IEICE TRANS. INF. & SYST., VOL.E106–D, NO.5 MAY 2023

PAPER

A Practical Model Driven Approach for Designing Security Aware RESTful Web APIs Using SOFL

Busalire Onesmus EMEKA[†*a)], ***Student Member***, **Soichiro HIDAKA**[†b)], ***and*** **Shaoying LIU**[††c)], ***Members***

SUMMARY RESTful web APIs have become ubiquitous with most modern web applications embracing the micro-service architecture. A RESTful API provides data over the network using HTTP probably interacting with databases and other services and must preserve its security properties. However, REST is not a protocol but rather a set of guidelines on how to design resources accessed over HTTP endpoints. There are guidelines on how related resources should be structured with hierarchical URIs as well as how the different HTTP verbs should be used to represent well-defined actions on those resources. Whereas security has always been critical in the design of RESTful APIs, there are few or no clear model driven engineering techniques utilizing a secure-by-design approach that interweaves both the functional and security requirements. We therefore propose an approach to specifying APIs functional and security requirements with the practical Structured-Object-oriented Formal Language (SOFL). Our proposed approach provides a generic methodology for designing security aware APIs by utilizing concepts of domain models, domain primitives, Ecore metamodel and SOFL. We also describe a case study to evaluate the effectiveness of our approach and discuss important issues in relation to the practical applicability of our method.

key words: *RESTful web APIs, SOFL, API security, secure by design, domain driven design*

1. Introduction

Nowadays, most of the web APIs are continuously adopting REpresentational State Transfer (REST) [1] architectural style which allows building loosely coupled API designs relying on HTTP and the web friendly JSON data representation format or XML. A web API (Application Programming Interface) is a set of functions and procedures that allow users to access and build upon the data and functionality of an existing application available over the web through the HTTP protocol [2]. The loosely coupling approach makes client applications have flexibility and reusability of an API in terms of the fact that its elements can be easily added, replaced, and changed. However, REST is a design paradigm and protocol-agnostic, an abstraction of the basic architecture of the HTTP protocol and concentrates on concepts rather than syntax and technical implementation details. It does not rely on any set of defined standards to describe the implementation of a RESTful API. Moreover, when it comes to security of RESTful APIs, the architectural style does not provide any built-in security capabilities. Security depends entirely on the design of the API itself. This means security must be actively built for data transmission, deployment, and interaction with clients. These limitations pose a challenge in the development and testing for satisfiability of a RESTful APIs' functional and security properties.

In our previous publications [3] we proposed a method offering a practical approach to specify and verify security and functional requirements of RESTful APIs using SOFL (Structured-Object-oriented Formal Language) [4]. In our approach, we focused on ensuring that all of the expected functional behaviors provided by an API and their related security requirements were captured correctly. To achieve this, we constructed a comprehensible SOFL functional and security requirement specifications from an API description written in RESTful API modeling language (RAML) [5].

However, this approach has a couple of shortcomings in practice. First, engineers need to think about APIs security vulnerability issues while at the same time focus on solving business functionality. This is difficult because, when engineers write specifications and code, mostly their main focus will be on the functionality they are trying to implement. Second, it requires every engineer to be well versed in API security issues, and finally it assumes engineers can think of any potential API vulnerability that may occur now and in the future. However, practically, a software engineer can only provide countermeasures for security vulnerabilities that he or she is only aware of. Furthermore, security is relative to a perceived threat model and an API can be more or less secure to that perceived threat model. To create secure APIs efficiently, one needs to have a mindset that focuses more on the design rather than on security, while at the same time pay attention to APIs' threat modeling techniques that can identify common API vulnerabilities and their recommended countermeasures. This makes security to be treated as a concern rather than a feature in API design and deployment. This approach has the benefit of providing a comprehensive approach of developing APIs that are not only protected from published API security vulnerabilities but also offers a potential of protecting an API from future mutations of existing API security vulnerabilities by just re-running the threat modeling that are adjusted to the mutated vulnerabilities.

Manuscript received November 4, 2022.
Manuscript revised January 1, 2023.
Manuscript publicized February 13, 2023.

[†]The authors are with Graduate School of Computer and Information Sciences, Hosei University, Koganei-shi, 184–8584 Japan.
[††]The author is with Graduate School of Advanced Science and Engineering, Hiroshima University, Higashihiroshima-shi, 739–0046 Japan.
[*]Presently with AlpsAlpine Co., Ltd, Iwaki, Fukushima, 970–1144 Japan.
a) E-mail: dr.b.emeka@gmail.com
b) E-mail: hidaka@hosei.ac.jp
c) E-mail: sliu@hiroshima-u.ac.jp
DOI: 10.1587/transinf.2022EDP7194

In this paper, we propose a new practical model-driven approach that addresses the shortcomings of our previous publications by proposing a methodology that allows for both explicit and implicit addressing of API security issues. We present a Domain Driven Design [6] approach that utilizes a metamodel offering a foundation for what an API does. The model is strict in the mathematical sense of being precise and exact that its concepts, attributes, behaviors and relations are unambiguous. The model relies on domain primitives [7] which combine secure constructs, and value objects to define the smallest building blocks of a domain, and utilizes Ecore [8] metamodel that defines the abstract syntax, the possible domain elements, and their relations in between. We adopt Attack-Defense Trees (ADTrees) [9] as our threat model of choice for identifying countermeasures to RESTful API security vulnerabilities. Our contribution lies on the overall methodology of providing a secure by design approach for writing specifications for RESTful APIs with semi-automation of some of its processes such as generation of a scaffold for API domain models from source RAML document input. Our proposed approach targets design and implementation of RESTful web APIs for web applications either internally developed by the same team or developed by a third party.

Specifically, we have made the following new contributions. We propose, a new model-driven approach for interweaving functional and security requirements based on Domain Driven Design principles, Ecore metamodel and the expressive nature of SOFL process definition to describe the behavior of our modeled API. This encourages resolving security issues both implicitly and explicitly. Implicitly by applying strict invariants on domain primitives [7], and explicitly by applying ADTrees [9] to model API threats and identify common documented API vulnerabilities and their associated countermeasures. By focusing on the domain and domain primitives, many security bugs can be solved implicitly. For example, applying a strict invariant defined as a domain primitive on an API's POST input not only protects the API endpoint against injection attacks but also ensures the specified meaning of the input is captured. Therefore, any malicious input not satisfying the definition is rejected and the API endpoint becomes more secure. We have demonstrated a practical usage of our approach by our implemented case study project containing most of the artifacts which is accessible via this url https://github.com/Egalaxykenya/IEICE-journal-paper-emeka.

The remainder of this paper is organized as follows. Section 2 describes briefly SOFL, ADTrees and REST architectural style. Section 3 describes our proposed Domain Driven Design model and metamodel that achieves both implicit and explicit interweaving of API's security and functional requirements. Section 4 describes an optional specifications testing activity aimed at verifying various consistency properties of the specifications generated via our proposed approach, and whether the specifications reflect the target API's requirements. Section 5 gives a brief discussion on the evaluation of our proposed approach. Section 6 focuses on related work and finally Sect. 7 gives conclusions and outlines areas for our future research.

2. SOFL, ADTrees, DDD, Metamodeling and REST - Background

In this section, we briefly introduce SOFL, ADTrees, Domain Driven Design, metamodeling concepts and REST in order to pave way for the discussion of our proposed approach.

2.1 SOFL

SOFL is a formal engineering method that provides a comprehensible language for both requirements and design specifications, and a practical method for developing software systems [10]. SOFL is designed by integrating different notations and techniques on the basis that all are needed to work together effectively in a coherent manner for specification construction and verification. The SOFL specification language integrates Conditional Data Flow Diagrams (CDFDs) [10] which describe comprehensibly the architecture of specifications, Petri nets [11] and VDM-SL (Vienna Development Method - Specification Language) [12]. SOFL also uses classes to model complicated data flows and stores. A data store offers data that can be accessed by processes in a CDFD. SOFL adopts a three-step evolutionary approach to developing formal specifications, starting from informal specifications to semi-formal specifications and finally to formal specifications. The informal specification usually written in a natural language serves as the basis for deriving the semi-formal specifications in which the SOFL syntax to some extent is enforced. The formal specification is then derived from the semi-formal specification through formalization of the informal parts in the semi-formal specifications. A SOFL formal specification consists of a group of *modules* organized in a hierarchical manner. Each module encapsulates the related *processes* that specify the expected functions, the *datastores* that specify the data resources accessed by the process, and the *invariants* that specify the constraints to be conformed by the process and data stores. A process is composed of five parts *name, input ports, output ports, pre-condition* and *post-condition*.

The input and output ports specify the input and output variables of the process. The pre- and post conditions define the semantics of the process interpreted as follows: When one of the input ports is available i.e. all of its input variables are bound to specific values in their types, the process will be executed. As a result of the execution, one of the output ports is made available, which means all of their respective output variables are bound to specific values of their types. If the input variables satisfy the pre-condition before the execution, the output variables are required to satisfy the post-condition after the execution of the process, provided the execution terminates. Listing 1 shows a typical SOFL formal specification and its corresponding CDFD is shown

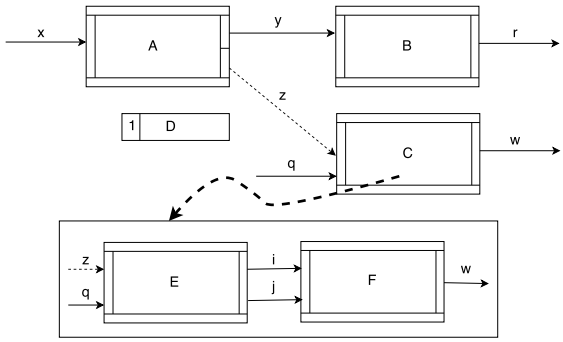

Fig. 1 An Example of a SOFL CDFD

in Fig. 1.

The SOFL formal specifications describe two modules i.e. *Arithmetic* and *Arithmetic_decom*. The module *Arithmetic_decom* is nested within the module *Arithmetic*. Each module contains a number of processes where each process describes an independent system function. Module *Arithmetic* consists of processes *A, B* and *C*. Process *C* is represented as the decomposed module *Arithmetic_decom* in Listing 1 lines 17 - 29. These three processes are connected by data flows, which represents the overall function of module *Arithmetic*. The decomposed lower level module *Arithmetic_decom* consisting of processes *E* and *F* represent process *C* in the parent module. An interpretation of the formal specifications of process *A* is as follows. It consists of one input port and two output ports separated by notation |. It takes *x* of integer type as the input variable and produces either *y* or *z* as the output variable. Its pre-condition is set to true and its post-condition requires that the output variable *y* is equal to the square of *x* if *x* is greater than 0, and the external variable *D* will be updated by the following condition *D* = ˜*D* + x; otherwise variable *z* will be made available. The ˜ sign before the variable *D* symbolises the initial value of the variable *D* before it is updated by the process *A*.

The semantics of SOFL CDFD are interpreted as follows: In each CDFD, a process is represented by a rectangle box with a name in the center. Each input port is denoted by a narrow rectangle on the left part of the process box, which receives input data flows. Each output port is denoted by a narrow rectangle to the right part of the process which produces output data flows. Multiple input and output ports are denoted by multiple narrow rectangles to the left and right parts of the process respectively. The pre- and post conditions are denoted by rectangles located in the upper and lower parts of the process. While using a supporting tool, a mouse click on these areas would give access to the pre- and post conditions respectively. In addition, CDFDs put focus of attention on data flow but not on control. Therefore, there is no explicit linking of input and output ports within a process [13].

More details about the SOFL specifications language can be found from the SOFL book [4].

```
1 module Arithmetic
2
3     process A(x: int) y: int | z: sign
4         ext #wr D: int
5
6         pre true
7         post (( x > 0 and D = ~D + x and y = x**2) or ( x <= 0 and bound(z))
8
9     end_process;
10
11    process B(y:int) r: int
12        pre y > 0
13        post r = y * 5
14    end_process;
15
16    /* Process C decomposed into lower level processes E and F */
17    module Arithmetic_decom
18
19        process E(z: sign, q: int) i:int, j: int
20            pre (q > 0 and bound(z))
21            post ( i = q + 1 and j = q ** 2)
22
23        end_process;
24
25        process F(i: int, j: int) w: int
26            pre ( i > 0 and j > 5)
27            post w = i * j
28        end_process;
29    end_module;
30 end_module;
```

Listing 1 SOFL formal specification example

2.2 ADTrees

Kordy et al. [9] define an ADTree as a node-labelled rooted tree describing the measures an attacker might take to attack a system and the defenses that a defender can employ to protect the system. ADTrees have nodes of two opposite types: attack nodes and defense nodes, which correspond to an attacker's and a defender's (sub-)goals, respectively. In their literature, they identify two key features of an ADTree; the representation of refinements and countermeasures. Every node of an ADTree may have one or more children of the same type representing a refinement into sub-goals of the node's goal. A node which does not have any children of the same type is called a non-refined node. Non-refined nodes represent basic actions in attack tree threat modelling.

Every node of an ADTree may also have a child of opposite type, that represents a defense. Therefore, an attack node may be represented by several children which refine the attack and another type of child which defends against the attack. The defending child in turn may have several children which refine the defense and one child, i.e., an attack node that counters the defense. A node of an ADTree can be refined disjunctively or conjunctively. The goal of a disjunctively refined node is achieved when at least one of its children's goals is achieved. The goal of a conjunctively refined node is achieved when all its children's goals are achieved. ADTrees core purpose is to model attack–defense scenarios. An attack–defense scenario can be seen as an interaction between a proponent and opponent. The root of an ADTree represents the main goal of the proponent. If the root is defined as an attack node, the proponent in this case will be an attacker and the opponent will

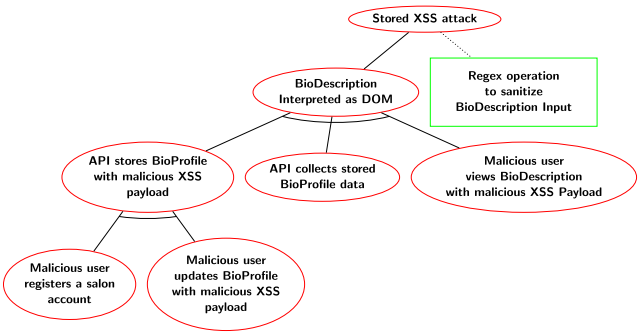

Fig. 2 Stored XSS ADTree threat modelling

be a defender. Conversely, if the root is defined as a defense node, the proponent in this case will be a defender and the opponent an attacker. Kordy et al. [9] propose the following graphical semantics when drawing ADTrees. Attack nodes are depicted by circles and defense nodes by rectangles, as shown in Fig. 2. Refinement relations are indicated by solid edges between nodes, and countermeasures are indicated by dotted edges. Conjunctive refinement of a node is depicted by an arc over all edges connecting the node and its children of equal type. To demonstrate an example of an ADTree, we consider a manifestation of stored Cross Site Scripting attack scenario at an API endpoint as shown in Fig. 2. Its root node is an attack, thus the main goal expressed by the tree are scenarios that can lead to execution of a successful stored XSS attack at any given API endpoint and their respective countermeasures. To launch a successful stored XSS attack against a web API endpoint, an attacker can choose multiple vectors of manipulating web API's endpoint accepting user inputs, by providing malicious input which is later interpreted as a Document Object Model (DOM) when loaded on a browser thereby executing the target attack. Assuming we are dealing with an API endpoint of a salon management system for updating a users bio profile as indicated in Fig. 2, the following sequence of events need to happen for a successful execution of a stored XSS attack. First, a malicious user A must register an account in the salon management system and must provide the salon web API endpoint for updating his bio profile with a malicious XSS payload. The API will then store the user's bio profile alongside the malicious XSS payload. Any user who will then access user A's bio profile via a web browser will make the uploaded malicious payload stored alongside user A's bio profile to be interpreted as a DOM thereby triggering a successful execution of a stored XSS attack. Therefore, a quick countermeasure against this kind of attacks is to sanitize input at any of the target API endpoints. One way of achieving this is to define a string processing function that strictly checks for potentially malicious characters that can be interpreted by the browser as DOM objects from our API payload. A sample specification implementation of this counter measure can be accessed in our case study github repository, lines 50 - 74 and 257 - 277 of *salon_api_sofl_implicit.txt* file. This countermeasure works towards preventing a successful execution of our attack tree root node goal. Compared to other threat modelling techniques, Attack-Defense Trees provide an intuitive graphical representation of different attacks which enable them bridge the gap between stakeholders coming from diverse backgrounds. This enables them to not only detect, analyze, brainstorm, amend results of an attack analysis and document a wide range of attacks but also define reactive countermeasures against the attacks. The stakeholders benefit by relying on a framework that provides a succinct and meaningful structure for a range of potential attack vectors in their system modeling activities.

2.3 Domain Driven Design(DDD)

Domain models provide unambiguous, strict foundation of what a system should do [6]. This by extension provides a powerful tool that defines what a system should not do. When modeling and implementing that model as requirements specifications, it is crucial to have some building blocks. Domain models are usually based on value objects and entities with larger structures being presented through aggregates. An aggregate is a conceptual boundary that you can use to group parts of the model together allowing you to treat an aggregate as a unit during state changes. The boundary is not arbitrarily chosen but rather it is carefully selected based on deep insights of the model. In order to express a domain model in specifications, you need a set of building blocks which are entities, value objects and aggregates.

Every part of a domain model has certain characteristics and certain meaning. Entities are model objects that have some distinct properties, unique identifiers and are responsible for coordination of the objects they own, not only to provide cohesion but also to maintain their internal invariants. Let's take an example of an Online Salon Booking System (OSBS), where we have a salon class with attributes such as salon name, address, phone number, establishment data. Every instance of entities has a unique identifier. The ability to identify information in a precise manner as well as coordinating and controlling behavior plays an integral role in preventing security bugs from sneaking into specifications. Thanks to this uniqueness, we can distinguish two instances of the salon class that has the same name, and even have all the same attribute values, by their identifiers, even if they can be interchangeable with each other. Entities are often made up of other model objects. Some attributes and behaviors can be moved out of the entity itself and put into other objects thereby becoming value objects or domain primitives. Value objects have no identity that defines them but rather, they are defined by their values, they are immutable i.e. they describe some attribute or some characteristics but carries no concept of identity, they can reference entities, they explicitly define and enforce important constraints and can be used as attributes of entities and other value objects. *Domain primitives* represents objects consisting of attributes and behaviors that were moved out of an entity and put into other objects. They are distilled versions

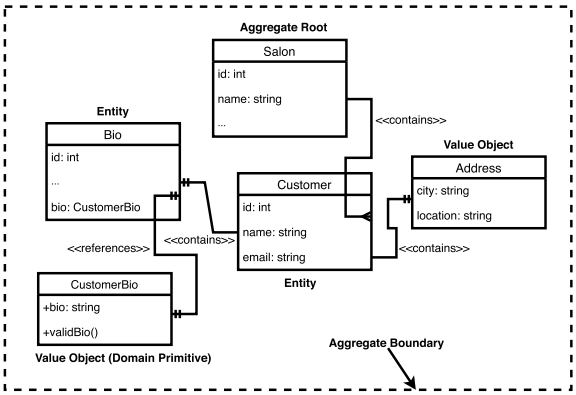

Fig. 3 Aggregate, entity, value objects & domain primitive in an example of a salon service booking system domain model

of value objects with proper invariants. These invariants enforce security constraints on the behavior of their associated entities in our API domain model. Figure 3 shows the relationships among aggregates, entities, value objects and domain primitives in DDD using our salon service booking system example. The entity *Salon* represents the aggregate root. The *Salon* entity has a containment relationship with the entity *Customer* which in turn has a containment relationship with both the entity *Bio* and a value object *Address*. The *CustomerBio* value object has a referential relationship with the entity *Bio*. It is a defense mechanism output of an ADTree analysis on the entity *Bio* that seeks to provide an invariant as a domain primitive. This invariant strictly whitelists a range of characters allowed as valid inputs for a customer's *Bio* description.

2.4 Metamodeling

A metamodel of a model specifies its structure and meaning. It defines the abstract syntax, the possible elements, and their relations in between. In addition, it specifies its static semantics, the constraints for well-formed models. There are two popular meta-metamodels. The Meta Object Facility (MOF) [14] by the Object Management Group (OMG) which is used as meta-metamodel for the Unified Modeling Language (UML) and Ecore [8] which is part of the Eclipse Modeling Framework (EMF) and based on Essential MOF (EMOF) [14]. We choose Ecore because it has more freely available supporting tool. We shall describe the needed Ecore elements and their properties next. There are four Ecore classes needed to represent a metamodel:

EClass Used to represent a modeled class. It has a name, zero or more attributes, and zero or more references. It can define a set of *EAttributes* describing its properties and a set of *EReferences* describing its relations to other *EClasses*. An *EClass* can be marked abstract which means an instance of itself cannot be initiated when created.

EAttribute Used to represent a modeled attribute. Attributes have a name and a type.

EDataType Used to represent the type of an attribute. A data type can be a primitive type like int, float or an object type.

EReference Used to represent one end of an association between classes. It has a name, a Boolean flag to indicate if it represents containment, and a reference (target) type, which is another class.

We use concepts of metamodeling in step 5 of our proposed approach.

2.5 REST and REST Concepts

The concept of REST was introduced by Roy Fielding in his PhD dissertation, *"Architectural Styles and the Design of Network-based Software Architectures"* [1]. REST relies on HTTP protocol for data communication and revolves around the concept of resources where each component is considered as a resource. These resources are accessed via a common interface using HTTP methods such as GET for retrieving a resource, PUT for updating a resource, POST for creating a resource and DELETE for removing a resource. Contrary to other web services, REST is an architectural style and protocol agnostic. The REST architecture focuses on providing access to a resource for a REST client to access and render it [1]. It utilizes Uniform Resource Identifiers (URIs) in identifying each resource and provides several resource representations such as XML, JSON, Text etc. to represent its type. For an API to be considered RESTful, it needs to satisfy the design characteristics commonly referred to as REST constraints [1] i.e., *Client-server architecture*, *Statelessness*, *Caching*, *Uniform Interface*, *Layered systems*, and/or *Code on Demand* (optional). A detailed description of RESTful web APIs is given in [2].

2.5.1 ResourceType and ResourceIdentifierPatterns

A *ResourceType* represents a RESTful API concept that models an object and its set of properties. It is defined as an abstract *EClass* with a name. It has an attribute *maxResources* which specifies the number of resources allowed. The uniform interface REST constraint dictates that activities which transcend create, read, update, and delete (CRUD) operations, must be modeled in a different way. For example, if we are creating an Online Salon Booking API with a capability for suggesting the best salon, a suitable workflow needs to be defined: salons can be suggested, and customers must share their reviews. Such a suggestion can be modeled as an *ActivityResourceType*. An *ActivityResourceType* is normally a nominalisation of an activity. A *ResourceIdentifierPattern* describes a URI to a *ResourceType*. Since we are specifically modeling RESTful APIs and by extension adhering to REST semantics, every *ResourceType* must have at least one *ResourceIdentifierPattern*. A *ResourceIdentifierPattern* is abstract and can be a *SimpleIdentifier* which is described using a string or a *ComplexIdentifierPattern* which uses the values of the *ResourceType's* Attributes. A *ResourceType* contains an unordered set of named *ResourceElements*, like attributes and links.

2.5.2 DataTypes and Attributes

Attributes specify a *ResourceType's* properties and conform to a defined *DataType*. A *DataType* can be a *PrimitiveDataType*, a *domain primitive* or a *CollectionType*. *PrimitiveDataTypes* are identified by their name, for example, integers and strings. A *CollectionType* represents an ordered set of values and references the *DataType* of its contained elements.

2.5.3 Method, MethodType and Parameter

REST-based services use HTTP interfaces, such as GET, PUT, POST and DELETE, to maintain uniformity across the web. The uniform interface, enforces a *MethodType*. A *MethodType* is identified by its name, to be defined for all existing methods, i.e., the HTTP verbs. A *ResourceType* is associated with a set of supported Methods which must have a *MethodType*. The Method element is responsible for the API behavior and determines the set of produced and consumed *MediaTypes*. In addition, every Method can define parameters which can be contained in a consumed *MediaType* or in the resource identifier.

2.5.4 Link and RelationType

Links support hypermedia as the engine of application state (HATEOAS) [1]. Each Link can define a media type independent RelationType [15]. This means the client is aware of the relation existing between two resources and which method requests with which meaning can be sent to the target link. A *RelationType* can contain pagination information like next or previous. An *InternalLink* refers to one target *ResourceType*. To model links to resources outside the current application *ExternalLinks* are used. These only go to the extent of defining resource identifiers.

3. Proposed Model-Driven Approach

3.1 Overview

Our proposed model-driven approach offers a 6 step process (Fig. 4) and focuses on interweaving APIs' functional and their respective security requirements. We adopt concepts of Domain Driven Design, Ecore metamodels and SOFL as the building blocks of our methodology. Domain Driven Design and Ecore metamodel enable our approach achieve structural modeling of an API while the expressive nature of SOFL and its definition of SOFL processes enables our approach to achieve behavioral modeling of an API. Our approach aims at resolving API security issues both implicitly and explicitly. Implicitly by applying strict invariants on a domain primitive, and explicitly by applying ADTrees to model API threats and vulnerabilities. We rely on official documented repositories such as OWASP API security top 10 [16] and Common Vulnerabilities Exposure (CVE) [17] database as a baseline to guide in identifying common documented RESTful API vulnerabilities. Figure 4 gives a diagrammatic representation of the general overview of our proposed model. In the following subsequent sections, we describe in detail the steps of our proposed approach and how they work.

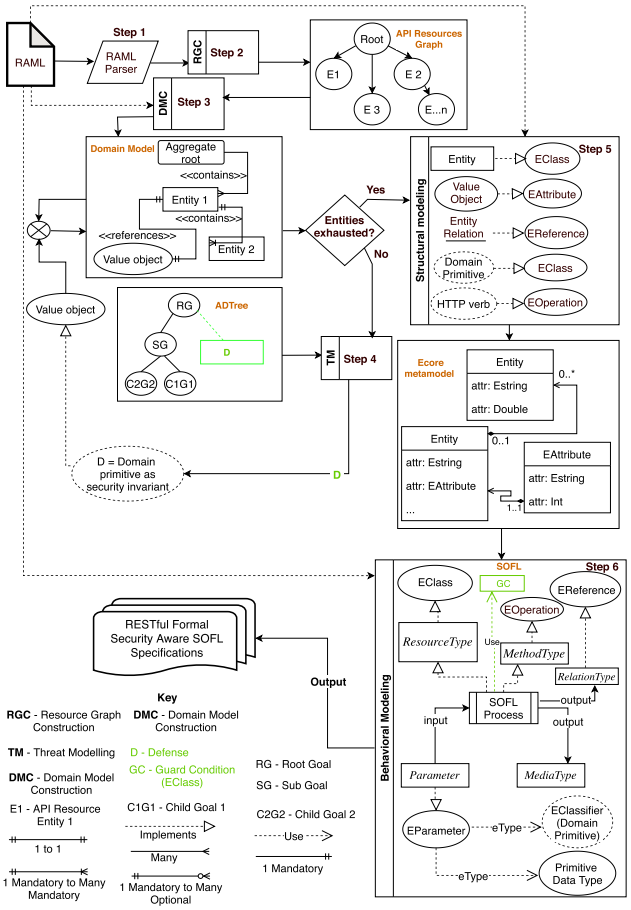

Fig. 4 Our proposed model. Steps 1 and 2 are automated while the rest of the steps require manual interaction

3.2 Step 1 - RAML Parsing

The first step involves generating a flat file with APIs resource listings. In this step, we parse a RESTful API documentation written in RAML with resource definitions as input into an RAML parser [18]. A resource in RAML is identified by its relative Uniform Resource Identifier (URI), which must begin with a slash ("/"). A resource may refer to other resources via steps ("/") in URIs. The resource may be a containment (child) node or otherwise referred to as a nested resource, or non-contained resource. Containedness is determined in a later step. A nested resource will have its URI relative to the parent resource URI. A sample of a flat file that is generated as an output of this step is given by Listing 2. This step is fully automated.

```
1 /salons
2 /salons/{salon_id}/service-categories
3 /salons/{salon_id}/service-categories/{category_id}
4 /salons/{salon_id}/service-categories/{category_id}/salon-services
5 /salons/{salon_id}/service-categories/{category_id}/salon-services/{
      service_id}
6 /salons/{salon_id}/service-categories/{category_id}/salon-services/{
      service_id}/bookings
7 /salons/{salon_id}/customers
8 /salons/{salon_id}/customers/{customer_id}
9 /salons/{salon_id}/customers/{customer_id}/bookings
10 /salons/{salon_id}/customers/{customer_id}/bookings/{booking_id}
11 /salons/{salon_id}/booking-payments
12 /salons/{salon_id}/customers/{customer_id}/bookings/{booking_id}/
      booking-payments
13 /salons/{salon_id}/stylists
14 /salons/{salon_id}/stylists/{stylist_id}/salon-services
15 /salons/{salon_id}/stylists/{stylist_id}/bookings
16 /salons/{salon_id}/users
```

Listing 2 Sample flat file with API resource listings

3.3 Step 2 - API Resource Graph Construction

The second step involves automatic construction of an API resource graph that will work as a blue print for creating the target API domain model. The input for this step is the flat file generated from step 1 and the output is a directed graph(digraph) of API resources. We utilize the algorithm defined in Algorithm 1 which takes a list of lists of API resource nodes and the defined API root resource node as an input, and constructs a digraph highlighting all the API resources as an output. For example, the following invocation ENTITYGRAPH *([[salons, service-categories], [salons, salon-services]], salons)* returns *({salons, service-categories, salon-services}, {(salons, service-categories), (salons, salon-services)})*. It is worth noting that in the implementation of the algorithm we had to conduct some preprocessing on the contents of the input flat file such as stripping of ("/") and {id} to extract the target API resources from resource URIs which adopt REST URIs pattern as depicted by a sample shown in Listing 2.

Figure 5 shows a sample API entity resource digraph automatically generated by an implementation of the algorithm using Listing 2 as a source input. Note the generated digraph abstracts containment and reference relationships between the digraph's resource nodes. A containment relationship describes a relation where a business object represented as a resource entity can contain one or more other business objects with the containing business object known as the parent object while the contained objects are referred to as child objects. A reference relationship describes a relationship between business objects that is not embedding i.e. when you query a business object, its referenced objects are not automatically returned like the case of containment relationships. Similar to step 1, this step is also fully automated.

3.4 Step 3 - Domain Model Construction

In step 3, we use the generated digraph as a guide to manually define the API's initial domain model as the target output with an aggregate root corresponding to the root node of the input digraph and the rest of the nodes corresponding to domain model entities. In actual sense, the generated digraph in step 2 is a barebone representation of the target domain model, but with missing in the domain model at this stage in the distinction between containment and reference relationship between entities. As we construct the domain model, we rely on the encoded business logic defined in the RAML specifications to explicitly define containment and

Algorithm 1 API entity resource graph

```
1: procedure ENTITYGRAPH(L, v):  ▷ L is a list of list of resource names, v is root resource name
2:     let W be an empty set of resource names
3:     let V be an empty set of nodes
4:     let S be an empty set of edges
5:     let M be an empty map from node names to nodes
6:     let r be the root node
7:     for all list l of resource names in L do  ▷ Extract a set of unique resource names
8:         for all resource names n in l do
9:             W ← W ∪ {n}
10:        end for
11:    end for
12:    num ← 1
13:    r ← new Node()  ▷ Node structure consists of name and order
14:    r.name ← v
15:    r.order ← num
16:    num ← num + 1
17:    M ← M ∪ {v ↦ r}
18:    for all resource names n in (W \ v) do
19:        x ← new Node()
20:        x.name ← n
21:        x.order ← num
22:        num ← num + 1
23:        M ← M ∪ {n ↦ x}
24:        V ← V ∪ {x}
25:    end for
26:    for all list l of resource names in L do▷ Extract set of unique edges
27:        for all pairs of adjacent resource names (m,n) in l do
28:            S ← S ∪ {(M[m], M[n])}
29:        end for
30:    end for
31:    return (V, S)
32: end procedure
```

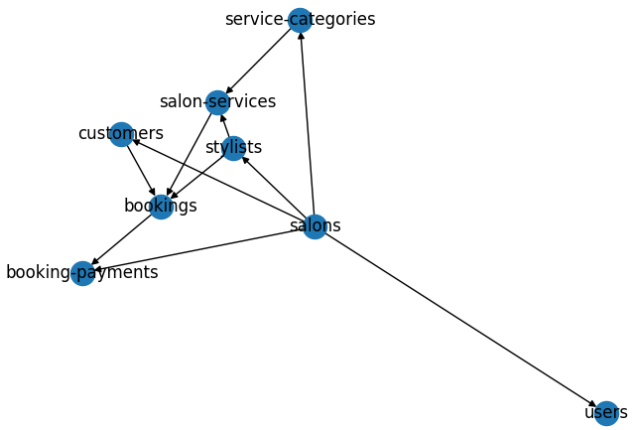

Fig. 5 API entity resource graph

reference relationships between domain model entities. We define the security schemes captured in the source RAML specifications as domain primitives in the constructed domain model. This encourages traceability between a domain model and the final generated SOFL specifications in step 6. The importance of an efficient domain layer is key to a successful secure by design API implementation. APIs developed based on the domain layer ensure businesses and security concerns gain equal priority in the view of both business experts and developers.

The defined domain model describes the entire ecosystem of the modeled API in the form of Domain Driven Design Concepts of aggregate root, entities, entity relationships and domain primitives.

3.5 Step 4 - Threat Modeling with ADTrees

The fourth step, which takes our newly defined domain model as input, involves a threat modeling process using ADTrees to identify potential security vulnerabilities in our API domain model and their countermeasures. Countermeasures that can enforce secure constructs on the attributes and behavior of their associated domain entities are modeled as domain primitives. The output of this step is a complete refined domain model with additional security invariants from the threat modeling process defined as domain primitives in the refined domain model. This fourth step achieves our first interweaving of functional and security requirements in an implicit manner. For countermeasures which involve consuming third party services such as rate limiting, request throttling and authentication, we flag them to be modelled as guard conditions and enforce their constraints later in the sixth step during behavioural modelling. The threat modelling process involves an analysis of all the domain entities for potential security vulnerabilities. The analysis process is a loop with the condition for proceeding to the next phase based on completion of exhaustive analysis of all the domain model entities representing the API resources. Through the ADTree analysis process, the domain model gets refined courtesy of new value objects being incorporated into the domain model as domain primitives.

3.6 Step 5 - API Structural Modeling

The fifth step involves creating an Ecore [8] metamodel that describes the structure of our API domain model. In this step, we rely on the refined domain model as input and create an Ecore metamodel that our refined domain model corresponds to, as an output. Specifically, this step encompass structural modeling of our target RESTful API. The structural model describes the possible resource types, their attributes, and relations as well as their interface and representations. We model *entities* as *EClasses*, *value objects* as *EAttributes* and their data type as *EDataType*, domain primitives as *EClass* and entity relations as *EReferences*. The modeling of aggregates is omitted since it's a collection of entities and value objects.

3.7 Step 6 - API Behavioral Modeling and SOFL Formal Specification Generation

The sixth and the final step involves behavioral modeling. The input for this step is an Ecore metamodel from step 5 and the output is formal security aware RESTful API specifications in SOFL language. Our goal here is to define RESTful API behaviors that consist of actions corresponding to their respective HTTP verbs i.e., GET, POST, PUT, DELETE and PATCH. For example, *CreateAction* creates a new resource, an *UpdateAction* provides the capability to change the value of attributes and *ReturnAction* allows for response definition including the Representation and all metadata. To achieve behavioral modelling, we transform our API methods into SOFL processes. A SOFL process definition is by itself a *MethodType* which takes inputs as *Parameters*, yields outputs of either *MediaType* or *RelationType* or both, defines a pre-condition and a post-condition, and can read or write a *ResourceType* to a data store. A *ResourceType* implements an *EClass* in our Ecore metamodel, *RelationType* implements an *EReference*, and a *Parameter* implements an *EParameter* which can have a type that is either a primitive data type, or a domain primitive which defines invariants that must be enforced at their point of creation. For example, an *UpdateAction* can be transformed into a SOFL process complete with pre-post conditions where the inputs are treated as the API's request parameters and outputs as the response including the Representation and all metadata. The semantic constraints of different *MethodTypes* are achieved naturally via the definitions of guard conditions in SOFL's post conditions. This step yields security aware formal RESTful API specifications as an output. While modelling the API behaviors as SOFL processes completed with pre-post conditions and guard conditions [4], our attention to security is drawn to:

- The resources exposed by the API that are to be protected
- Data transfer across the API's trust boundaries and aggregate boundaries
- The security goals that are important such as confidentiality of API resources
- The mechanisms that are available to achieve these goals such as authentication, access control, audit logging and rate limiting

To guide the formalization process of RESTful API behavior in SOFL, we first define the following SOFL formalization techniques with regards to a RESTful API. A RESTful API service D is defined as a set of operations $D = (o_1, o_2, \ldots, o_n)$ where $n \geq 1$. Each operation o_i $(1 \leq i \leq n)$ is represented by a pair of input and output messages in the format $o_i = (inMsg_i, outMsg_i)$. Each input message in $inMsg_i$ is defined as a set of input variables in the format $inMsg_i = \{v_1, \ldots, v_j\}$ $(j \geq 1)$. Similarly, each output message $outMsg_i$ is defined as $outMsg_i = \{v_1, \ldots, v_k\}$ $(k \geq 1)$. The potential functional behaviors are inferred from the re-

sources method definitions in the RAML source file which are encoded as *EOperations* in our API structural model. RESTful services represent business processes which may be organized in a hierarchical manner to represent business goals. Therefore each function can be further decomposed into low-level business processes. The formal representation of a REST service process in SOFL therefore involves 2 steps:

- Modularize REST service associated functions into proper SOFL processes. We adopt the following two rules during the modularization process:
  - Rule 1: If a function $F = \{f_1, \ldots, f_m\}$ is associated with service S, we construct a process P_i for each sub-function f_i $(i = 1, \ldots, m)$ of F.
  - Rule 2: If a function F requires a stateful data item x, then we construct a data store d to represent x. A datastore d specifies the expected data resource accessed by function F. It represents a necessary stateful variable that is shared by several processes.
- Fully formalize the pre- and post-conditions of these processes to precisely express the expected operational semantics upon their associated services.

We formaly define a SOFL process as a five-tuple: $(P, InPortSet, OutPortSet, preP, postP)$

- P is the name of the process
- $InPortSet = \{inPort_1, inPort_2, \ldots, inPort_f\}$ defines the set of input ports of P where $inPort_i$ $(i = 1, \ldots, f)$ is an input port. Each input port is defined as $inPort_i = \{v_{j_1}, \ldots, v_{j_{r_i}}\}$ where $v_k (k \in \{j_1, \ldots, j_{r_i}\})$ is a variable of this port.
- $OutPortSet = \{outPort_1, outPort_2, \ldots, outPort_g\}$ defines the set of output ports of P where $outPort_i$ $(i = 1, \ldots, g)$ is an output port. Each output port is defined as $outPort_i = \{v_{l_1}, \ldots, v_{l_{s_i}}\}$ where v_k $(k = l_1, \ldots, l_{s_i})$ is a variable of this port.
- $preP$ is the pre-condition of P, which specifies the condition that the input variables need to satisfy.
- $postP$ is the post-condition of P, which specifies the condition that the output variables are required to satisfy.

The semantics of a process P with respect to input and output ports corresponds to the interpretation of a CDFD diagram as described in Sect. 2.1 i.e. when one of the input ports in $InPortSet$, say $inPort_i$, is available, it means that all of its input variables are bound to specific values of their types and the process P will be executed. As a result of the execution, one of the output ports in $OutPortSet$, say $outPort_j$, is made available, meaning all of its output variables are bound to specific values of their types. If the input variables satisfy the pre-condition $preP$ before the execution of P, the output variables are required to satisfy the post-condition $postP$ after the execution of the process P, provided that the execution terminates.

To interweave security requirements with functional requirements at this stage, we introduce the concept of SOFL process functional scenarios [19]. The pre- and post-conditions of a SOFL process can be transformed into a number of independent relations called functional scenarios. Let the post-condition $P_{\text{post}} \equiv (C_1 \wedge D_1) \vee (C_2 \wedge D_2) \vee \ldots \vee (C_n \wedge D_n)$, where each C_i $(i = 1, \ldots, n)$ is a predicate called guard condition that contains neither output variables nor output external variables of the SOFL process and D_i is a predicate called defining condition that contains at least one output variable but does not contain any guard condition as its constituent expression [20]. Then each $\tilde{P}_{\text{pre}} \wedge C_i \wedge D_i$ is called a functional scenario, where $\tilde{F}$ for logical formula F of the input/output variables of a process denotes the value of F before starting execution of the process. The pre- and post conditions of a process P can then be transformed into a functional scenario of the form $\equiv (\tilde{P}_{\text{pre}} \wedge C_1 \wedge D_1) \vee \ldots \vee (\tilde{P}_{\text{pre}} \wedge C_n \wedge D_n)$. Each functional scenario $(\tilde{P}_{\text{pre}} \wedge C_i \wedge D_i)$ independently defines how the output of P is defined using D_i under the condition $\tilde{P}_{\text{pre}} \wedge C_i$. Guard conditions enforce invariants that constrain the behavior of their associated processes. In our case we define guard conditions that enforce security constraints thereby achieving the second interweaving of API's functional and security requirements.

4. Specification Testing

To verify whether the interweaved functional and security requirements implement all expected functions correctly and satisfy the desired security constraints, we can optionally perform specification testing to verify whether the specifications reflect the user requirements. Given a process $P \equiv (\tilde{P}_{\text{pre}} \wedge C_1 \wedge D_1) \vee \ldots \vee (\tilde{P}_{\text{pre}} \wedge C_n \wedge D_n)$ where $n \geq 1$, if we define a test set T, then T is said to satisfy the scenario-coverage of P if and only if $\forall_{i \in \{1,\ldots,n\}} \exists_{t \in T} \cdot P_{\text{pre}}(t) \wedge C_i(t)$. We interpret this as: A test set T satisfies the scenario coverage for the process P if and only if for any functional scenario, there exists a test case in T such that it satisfies the conjunction of the pre-condition $\tilde{P}_{\text{pre}}$ and the guard condition C_i. The test set T ensures that every functional scenario with its associated security constraint is covered appropriately.

To check for conformance of a process P specifications relative to user requirements of an API service operation o, we generate a test case t for each functional scenario $f_i \equiv \tilde{P}_{\text{pre}} \wedge C_i \wedge D_i$ using concrete input values and analyze the test results in order to determine whether violations of security constraints are detected. If r is the result of an API service operation o indicated by user specification using a test case t, and r' is the animation [21] (explained shortly) result of a process specification P using a test case t, if r' of the process P matches r, then we can confirm that process P property of operation o represents the users requirements and its associated constraints.

Implicit SOFL specifications do not indicate algorithms for implementations. However, they are expressed with predicate expressions involving pre and post conditions

Table 1 Testing RESTful service operation AddSalon.

Functional Scenario	Test Case	Execution Result	Expected Result
(access_token = validtoken ***and*** *access_token <> Nil* ***and*** *len(access_token) <> 0)* ***and*** *salons_table = conc(˜salons_table, salon)* ***and*** *response_message = "HTTP 200"*	(validtoken, "xvfKJgT", salon)	"HTTP 200"	"HTTP 200"
access_token <> validtoken **and** *elems(access_token) = {}* ***and*** *salons_table = ˜salons_table* ***and*** *response_message = "HTTP 401"*	(validtoken, " ", salon)	"HTTP 401"	"HTTP 401"
access_token <> validtoken ***and*** *access_token = Nil* ***and*** *salons_table = ˜salons_table* ***and*** *response_message = "HTTP 401"*	(validtoken, Nil, salon)	"HTTP 401"	"HTTP 401"

for a process and can be evaluated if all variables involved are substituted with concrete values of their types with results of such evaluations being truth values *true* or *false* [4]. For our specification testing, we further apply process animation technique to obtain the set of concrete values of output variables for each functional scenario. An analysis of a test results is done by comparing evaluation results with the analysis criteria. The analysis criteria is a predicate expression representing the properties to be verified. If the evaluation results are consistent with the predicate expression, the analysis show consistency between the process specification and its associated requirement. We generate the test cases for both input and output variables based on the user requirements. A simple running example can be used to demonstrate how we conduct specification testing to test if the specifications meet its critical requirements and also provides the desired functionality. In our case study, a RESTful API request operation *AddSalon* for creating a salon object needs to be checked on whether it satisfies its interweaved security requirement that requires an access token i.e. *guard condition* to be provided for a successful authorization for creation of a salon object. The required function on this operation is formally given as indicated in Listing 3. The process *AddSalon* takes a *validtoken*, an *access_token* and *salon* as input variables and returns an appropriate HTTP *response_message* as an output variable. The *validtoken* is a string constant used to verify the validity of the provided *access_token* by returning a boolean value. If we examine the pre-post conditions of the process *AddSalon*, we get three functional scenarios as follows:

(1) *(access_token=validtoken* ***and*** *access_token <> Nil* ***and*** *len(access_token) <> 0)* ***and*** *salons_table = conc(˜salons_table, [salon])* ***and*** *response_message = "HTTP 200"*

(2) *access_token <> validtoken* **and** *elems(access_token) = {}* ***and*** *salons_table = ˜salons_table* ***and*** *response_message = "HTTP 401"*

(3) *access_token <> validtoken* **and** *access_token = Nil* ***and*** *salons_table = ˜salons_table* ***and*** *response_message = "HTTP 401"*

We can generate test data from each functional scenario through specification animation as earlier described. Table 1 shows sample test cases covering the three functional scenarios and their corresponding results. The test cases generated are usually based on test targets which are predicate expressions, such as the pre and post conditions of a process. To cover for functional scenario (1), we provided a test case *(validtoken, "xvfKjT", salon)* for the required input variables *validtoken*, *access_token* and *sa-*

```
1 module Salon_API;
2 ...
3 token = string;
4 SalonData = composed of
5               id = string
6               owner = SalonUser
7               business_name = string
8               business_type = string
9               business_description = string
10              business_phone_number = string
11              business_email = string
12              business_address = AddressData
13              price_range = string
14              created = Timestamp
15            end;
16 SalonTable = seq of SalonData;
17 var
18 salons_table: SalonTable
19
20 inv
21 forall[i,j: inds(salons_table)] | i <> j
22            => salons_table(i).id <> salons_table(j).id;
23
24 process AddSalon(validtoken:token, access_token:token, salon: SalonData)
25                 response_message: string
26     ext wr salons_table
27     /* Pre condition: id of new salon must be unique */
28     pre not exists[i:inds(salon_table)] | salons_table(i).id = salon.id
29     post (access_token = validtoken
30         and access_token <> Nil
31         and len(access_token) <> 0)
32         and salons_table = conc(˜salons_table, [salon])
33         and response_message = "HTTP 200"
34         or salons_table = ˜salons_table and elems(access_token) = {}
35         and access_token <> validtoken
36         and response_message = "HTTP 401"
37         or salons_table = ˜salons_table and access_token = Nil
38         and access_token <> validtoken
39         and response_message = "HTTP 401"
40
41 end_process;
42 ...
```

Listing 3 SOFL formal specification for RESTful API AddSalon

lon object respectively. After executing the test case on a process specification *AddSalon*, the output value of variable *response_message* is equal to the expected output value inferred from the defining condition *response_message = "HTTP 200"*. For functional scenario (2), we run the test case *(validtoken, " ", salon)* and the value of the output variable *response_message* corresponds to the expected results. Running the test case *(validtoken, Nil, salon)* for functional scenario (3) also yields a value for the output variable that corresponds to the expected results as per interpretation of the user requirements. Therefore, we can determine that the process specification *AddSalon* does satisfy its critical requirements and as per its user requirements. Since our focus is on the relationship between input and output variables, and security concern rather than function, to simplify our presentation, the current test does not inspect data stores. For example, if empty access token is given to *AddSalon*, then *salon_datastore* should be untouched. In addition, the *validtoken* which could be provided by a third party service such as an authenticating service, is assumed to be always valid. However, these specifications are not explicitly captured in the test. We could incorporate data stores by extending our test cases. It is also worth noting that when testing for conformance of a process specification to its associated service operation, we only need to observe the execution results of the process by providing concrete input values to all of its functional scenarios analyzing their defining conditions relative to user requirements.

5. Evaluation of Our Proposed Approach

To evaluate the effectiveness of our proposed approach, we applied our methodology to an empirical case study of a service based on an *Online Salon Booking System (OSBS)*. We first created RAML specifications that defined our OSBS service. The specifications formed a foundation from which we built a domain model representing our OSBS. Listing 4 shows an excerpt of a sample RAML description of our OSBS API. For brevity, we do not showcase all the elements that went into building our domain model but rather focus on showcasing the general structure of an RAML description file. See our GitHub repository[†] for the full listing. Lines 7-14 show resources externally defined in separate files e.g. API data shapes, resource types and the authenticating security scheme. Lines 16-55 show sample resource type's endpoints and their associated HTTP interfaces with their corresponding request and response data shapes.

We then modeled our domain entities which were salons, stylists, services and customers as *PrimaryResourceTypes*. Every salon, service, stylist, and customer has a name and relevant *Attributes*. We defined the attributes of each entity in our domain as value objects. For example, the salon entity had attributes such as *salon_id*, *name*, *phone*, *email*, and *address*. A customer entity had attributes such as *customer_id*, *name*, *phone* and *bio_description*. Next, we iteratively conducted threat modeling activities using ADTrees on our domain model entities. Our goal here was to identify potential vulnerabilities and attacks that could be leveraged on our API resources and their appropriate countermeasures. We defined the countermeasures as security requirements which we implicitly or explicitly interweave with their related functional requirement. For example, the *bio_description* attribute could have been assigned a string data type directly. However, upon using ADTree to model threats on an API function that would persist a customer entity data shape into our database, we identified that if the *bio_description* were left to accept any string data type, it could make its associated API endpoint vulnerable to stored

```
1 #% RAML 1.0
2 title: SalonService API
3 baseUri: http://localhost:8000/api/{version}
4 version: v1
5 mediaType: application/json
6
7 uses:
8 shapes: ./dataTypes/shapes.raml
9
10 resourceTypes:
11 collection: !include resourceTypes/collection.raml
12
13 securitySchemes:
14 oauth_2_0: !include securitySchemes/oauth2_0.raml
15
16 /salons:
17    type:
18     collection:
19        response-type: shapes.SalonData[]
20        request-type: shapes.NewSalonRequestData
21    get:
22       description: Get a list of Salons based on the salon name
23       queryParameters:
24         salon_name:
25           displayName: Salon Name
26           type: string
27           description: Salon's name
28           example: "Salon Paradise"
29           required: true
30         ...
31
32    post:
33       description: Salon data created correctly for salon business
34       body: shapes.NewSalonRequestData
35    delete:
36     ...
37    /{salon_id}:
38       type: ...
39       get:
40        description: Get the salon with 'salon_id = {salon_id}'
41         responses:
42          200:
43           body:
44            application/json:
45             example: |
46              {
47               "data": {
48               "id": "lsVx",
49               "name": "Salon Paradise",
50               "location": "Chuo Ku,Tokyo",
51               "link":"http://localhost:8000/api/v1/salons/SalonParadise"
52             },
53             "success": true,
54             "status": 200
55            }
56              ...
```

Listing 4 Sample RAML description file

[†]Full case study: https://github.com/Egalaxykenya/IEICE-journal-paper-emeka

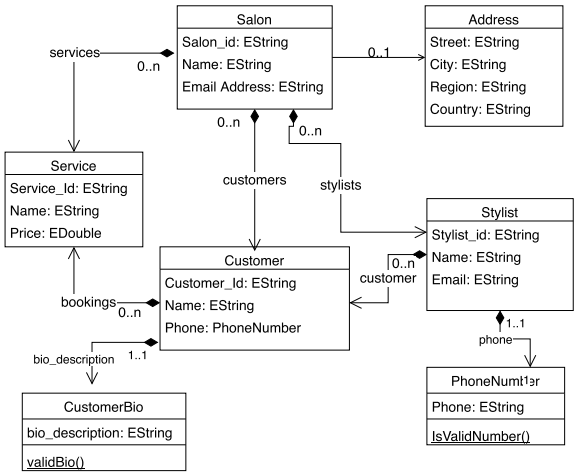

Fig. 6 Sample Ecore metamodel for OSBS

XSS [22] attack. An attacker can include malicious script as part of the *bio_description* which would later be injected into a browser's DOM [22] and wreak havoc, whenever the affected customer entity data shape is retrieved by users with elevated privileges in our OSBS. By strictly defining a valid string representation of a *bio_description* with an invariant such as a custom string processing function that checks for any script tags or advanced XSS payloads not written in plain text such as base64 or binary, we can protect our API endpoint from stored XSS vulnerability.

We achieved this by moving *bio_description* attribute out of the customer entity and created a domain primitive value object *CustomerBio*. This implemented an invariant that protected our customer entity data shape from stored XSS vulnerability and its potential future mutations. Moreover, it enforced validation checks that could assert the validity of the *bio_description* value object for a certain operation while at the same time making it possible for our OSBS to perform any other specific action on it. This technique whenever applied in our threat modeling process achieved implicit interweaving of functional and security requirements. Some of the identified countermeasures constrained the behavior of our API end points without the need for defining them as domain primitives. These countermeasures whenever incorporated in our modelling process achieve the explicit interweaving of functional and security requirements. For example, to protect our OSBS endpoints that implement PUT, PATCH, POST and DELETE *MethodTypes*, from Cross Site Request Forgery attacks [23], we could provide a unique token among other required parameters for each API request. We enforced such type of constraints later when doing behavioral modeling using SOFL. We then created a metamodel representing our OSBS domain model by transforming our entities to *EClasses*, value objects and domain primitives to *EDataTypes*, and entity relations to *EReferences*. Figure 6 shows a sample generated Ecore metamodel of our OSBS even though we only showcase a few modeled entities for brevity and simplicity purposes.

```
1  module Salon_API;
2
3      class Address;
4
5          var
6          street, city, region, country: string
7
8          method Init()
9
10             post street = "" and city = "" and country = ""
11                 and region = ""
12
13         end_method;
14
15     end_class;
16
17     type
18
19     SalonData = composed of
20                         salon_id = string
21                         name: string
22                         email: string
23                         address: Address
24                 end;
25     SalonCollection = seq of SalonData;
26     var salon_datastore: SalonCollection
27
28     inv /* salon_id uniquely identifies SalonData in salon_datastore */
29     forall[i,j: inds(salon_datastore)] | i <> j
30         => salon_datastore(i).salon_id
31           <> salon_datastore(j).salon_id;
32
33     process Search(search_id: string) salon_object: SalonData
34         ext rd salon_datastore
35         pre exists([i: inds(salon_datastore)] |
36                 salon_datastore(i).salon_id = search_id)
37         post salon_object inset elems(salon_datastore) and salon_object = search_id
38     end_process;
39
40     process UpdateSalon(salon_object: SalonData, token: string,
41                         salon_name: string) status_message: string
42         ext wr salon_datastore
43         pre exists([i: inds(salon_datastore)] |
44                 salon_datastore(i) = salon_object)
45         post len(salon_datastore) = len(~salon_datastore)
46             and (forall[i:inds(~salon_datastore)] |
47                 (~salon_datastore(i) = salon_object
48             => salon_datastore(i)
49                 = modify(~salon_datastore(i), name-->salon_name))
50               and (~salon_datastore(i) <> salon_object
51             => salon_datastore(i) = ~salon_datastore(i)))
52         and token <> "" and status_message = "HTTP 204"
53     end_process;
54 end_module;
```

Listing 5 SOFL formal specification for RESTful API UpdateAction

Finally, we generated formalized RESTful API specification via behavioral modeling of our designed metamodel. Here we transformed all the RESTful API methods for accessing the different *ResourceTypes* and *Collections* into SOFL processes with SOFL formal notation. A *Collection* represents a set of *ResourceTypes* exposed as XML or JSON. Listing 5 shows an excerpt of the end result of the API's SOFL formal specifications. For example, a RESTful API *UpdateAction* that updates the salon name of a single salon is transformed to a SOFL process *UpdateSalon* (lines 40-53), corresponding to PUT HTTP verb. The process relies on the output of a *Search* process that yields a *salon_object*. It then takes the *salon_object*, *new_salon_name* and a unique *token* as inputs, updates the salon's object attribute name and generates a status code that highlight suc-

Table 2 Summary of case study process specifications

No.	Process Specification	Functional Scenarios	Test Cases	Passed Test Cases
1	*RetrieveSalon*	4	4	3
2	*AddSalon*	4	4	3
3	*DeleteSalon*	4	4	3
4	*GetSalonService*	4	4	3
5	*DeleteSalonService*	4	4	3
6	*GetSalonCustomer*	4	4	3
7	*AddSalonCustomer*	4	4	3
8	*DeleteSalonCustomer*	4	4	3
9	*GetSalonBookings*	4	4	3
10	*GetSalonBooking*	4	4	3
11	*DeleteSalonBooking*	4	4	3
12	*AddSalonServiceCategory*	3	3	3
13	*GetSalonServiceCategories*	3	3	3
14	*GetSalonServiceCategory*	3	3	3
15	*DeleteSalonServiceCategory*	3	3	3
16	*GetSaloncategoryServices*	3	3	3
17	*CreateSalonService*	3	3	3
18	*RetrieveSalons*	3	3	3
19	*GetSalonCustomers*	3	3	3

cess or failure of the process operation. The unique token represents a security requirement that constrains the behavior of the *UpdateSalon* process as a guard condition (line 52). SOFL offers its own type declarations therefore, the salon *ResourceType* (object) will be declared as of a *composite* type, *salon_id*, *new_salon_name* and unique *token* parameters as of a *string* type and the data store declared as *sequence* type to represent a *Collection* of salon *ResourceTypes*. In SOFL notation, a data store variable with a tilde sign e.g. *˜salondatastore(i)* denotes the value of the data store before it is updated by a SOFL process. Table 2 gives a summary of a subset of all the process specifications of our case study, their number of functional scenarios and the test cases run for the functional scenarios. Rows 1 - 11 include process specifications which our proposed approach injected an access control security requirement to prevent Broken Object Level Authorization API vulnerability. A test relative to original RAML specification fails in the case where injected security measure (like requirement of an object level access control) is not respected, i.e., object level access control is not checked. Our generated SOFL specification correctly rejects such case (i.e., error message is returned), while the original RAML specification (incorrectly) dictates to accept such request, because it is not aware of such measure. A complete listing of all the process specifications with their functional scenarios can be accessed via this[†] full case study repository in the file *SOFL/salon_api_sofl_PFS.txt*.

5.1 Limitations

Through the application of our domain driven approach for modeling RESTful web APIs via a case study we have been able to demonstrate how our proposed approach can offer a comprehensive formalized approach for designing RESTful APIs. However, since we adopt SOFL formalism, this may impose a learning curve on engineers as they will need to learn and be conversant with SOFL semantics. However, in the long run, once they are conversant with SOFL, they will have the benefit of creating precise security aware API specifications using our approach. Further more, for most RESTful API projects, requirements are not fully formalized, rather, they are described in certain informal manner such as RAML or Swagger [24] and then translated into executable code due to external constraints such as budgetary limitations or time constraints. In such cases, the formalisation requirement may not be preferred by some project stakeholders. In addition, even though we rely on official documented databases such as OWASP and CVE to enable our threat modeling activity in our proposed approach to provide a wide coverage on API vulnerabilities, we cannot guarantee that all security vulnerabilities and threats will be addressed absolutely. However, since the threat modeling activities in our proposed approach encourage refinement through iteration, we are confident that our approach will provide a satisfactory coverage against most API security threats and vulnerabilities.

In spite of the aforementioned limitations, the application of our proposed approach in modeling security aware RESTful APIs in an implemented project has given us the confidence that it can help engineers have a blueprint for improving RESTful APIs requirement specifications in practice.

6. Related Works

As far as the security and modeling of web API's is concerned, several approaches have been done in the field of developing RESTful applications, but to the best of our knowledge, there are a few results that provide detailed model driven techniques with a focus on paying attention to both

[†]Full case study: https://github.com/Egalaxykenya/IEICE-journal-paper-emeka

APIs functional and security requirements at the same time. Fett et al. [25] propose a rigorous, systematic formal analysis of OpenID Financial-grade API (FAPI) based on a web infrastructure model. They first develop a precise model of the FAPI in the web infrastructure model, including different profiles for read-only and read-write access, different types of clients, and different combinations of security features, and use their model of FAPI to precisely define central security properties of an API. However, their model treat API security and functional requirements independently. Kopecky et al. [26] present hRESTS as a promising solution for providing a microformat model for RESTful services. However, their approach focuses more on documentation and service discovery with limited clarity on the relationship between API's security and functional requirements and their interdependence. Alqahatni et al. [27] introduce an approach for automatically tracing source code vulnerabilities at the API level across project boundaries. Their approach takes advantage of Semantic Web and its technology stack to establish a unified knowledge representation that can link and analyze vulnerabilities across project boundaries. However, they focus at the source code level rather than the design level. Klien et al. [28] provide an approach for showcasing how the constraints of REST and RESTful HTTP can be precisely formulated within temporal logic. However, their focus is mainly on formal characterization of REST for automated analysis.

Compared with the aforementioned related works and with our previous work as mentioned in Sect. 1, which also focused on interweaving APIs' functional and security requirements, our new approach provides a formal requirement specification methodology with an added advantage of providing a model driven secure by design approach for RESTful APIs. This encourages writing precise formal specifications defined by a metamodel. We take advantage of Domain Driven Design concepts, Attack-Defense Trees, Ecore metamodel and SOFL to provide a firm foundation for structural and behavioral modelling of APIs. In addition, our methodology lays a foundation for further running formal proofs and techniques such as specification testing on an API's specifications to verify its satisfiability of both functional and security requirements.

7. Conclusion and Future Works

In this paper, we have proposed a new six step modeling approach that encourages security aware API design principles. We present a Domain Driven Design approach for secure RESTful API design, that utilizes a metamodel offering a strict foundation for what an API does. The model relies on domain primitives which combine secure constructs and value objects to define the smallest building blocks of a domain and utilizes Ecore metamodel to define the abstract syntax, the possible domain elements, and the relations between them. We adopt Attack-Defense Trees as our threat model of choice for identifying documented API vulnerabilities and their respective countermeasures. To support the efficient use of our proposed model, we plan to enrich our model by incorporating an optional Object Constraint Language (OCL) support in our model as well as build a supporting tool in the future for semi-automating the transformation of generated formal API specifications into executable code.

Acknowledgements

This research was supported by ROIS NII Open Collaborative Research 2021-(21FS02) and JST, CREST Grant Number JPMJCR21M4, Japan. We would like to thank professor Toshio Hirotsu and professor Satoshi Obana, both from Hosei University, Faculty of Computer Science for their expertise and assistance in providing expert review of the manuscript. We would also like to thank the anonymous reviewers for their constructive comments and suggestions to improve the quality of our paper.

References

[1] R.T. Fielding, "Architectural styles and the design of network-based software architectures - Ph.D. dissertation," 2000.
[2] M.A.L. Richardson and S. Ruby, RESTful Web APIs, first ed., O'reilly Media Inc., Sebastopol, CA, Sept. 2013.
[3] B. Emeka, S. Hidaka, and S. Liu, "A formal approach to secure design of RESTful web APIs using SOFL," Structured Object-Oriented Formal Language and Method, ed. J. Xue, F. Nagoya, S. Liu, and Z. Duan, Cham, vol.12723, pp.105–125, Springer International Publishing, 2021.
[4] S. Liu, Formal Engineering for Industrial Software Development Using the SOFL Method, Springer-Verlag, Berlin, Heidelberg, New York, 2004.
[5] B. De, API Management: An Architect's Guide to Developing and Managing APIs for Your Organization, Apress, 2017.
[6] E. Evans, M. Fowler, and E. Evans, Domain-Driven Design: Tackling Complexity in the Heart of Software, Addison-Wesley, 2004.
[7] D. Sawano, D. Johnsson, and D. Deogun, "Domain primitives" in Secure by Design, Manning, 2019.
[8] D. Steinberg, ed., EMF: Eclipse Modeling Framework, 2nd ed., rev. and updated ed., The eclipse series, Addison-Wesley, Upper Saddle River, NJ, 2009. OCLC: ocn182662768.
[9] B. Kordy, S. Mauw, S. Radomirović, and P. Schweitzer, "Attack-defense trees," Journal of Logic and Computation, vol.24, no.1, pp.55–87, 2014.
[10] S. Liu, A.J. Offutt, C. Ho-Stuart, Y. Sun, and M. Ohba, "SOFL: a formal engineering methodology for industrial applications," IEEE Trans. Softw. Eng., vol.24, no.1, pp.24–45, Jan. 1998.
[11] M. Silva, Introducing Petri Nets, pp.1–62, Springer Netherlands, Dordrecht, 1993.
[12] C.B. Jones, Systematic Software Development Using VDM, 2nd edition ed., Prentice Hall, 1990.
[13] C. Ho-Stuart and S. Liu, "A formal operational semantics for SOFL," Proc. Joint 4th International Computer Science Conference and 4th Asia Pacific Software Engineering Conference, pp.52–61, 1997.
[14] Meta Object Facility (MOF) Core Specification, Object Management Group, Jan. 2006.
[15] M. Nottingham, "Web linking," Tech. Rep. IETF RFC 5988, Oct. 2010.
[16] "Owasp api security project | OWASP foundation," https://owasp.org/www-project-api-security/ (Accessed on Nov. 11. 2022).
[17] "Common vulnerability enumeration - CVE," https://cve.mitre.org/ (Accessed on Dec. 21. 2022).
[18] "An RAML parser based on SnakeYAML written in java,"

https://github.com/raml-org/raml-java-parser (Accessed on Aug. 09. 2022).

[19] S. Liu and S. Nakajima, "A decompositional approach to automatic test case generation based on formal specifications," 2010 Fourth International Conference on Secure Software Integration and Reliability Improvement, pp.147–155, 2010.

[20] S. Liu, F. Nagoya, Y. Chen, M. Goya, and J.A. McDermid, "An automated approach to specification-based program inspection," Formal Methods and Software Engineering, ed. K.K. Lau and R. Banach, Berlin, Heidelberg, vol.3785, pp.421–434, Springer Berlin, Heidelberg, 2005.

[21] M. Li and S. Liu, "Integrating animation-based inspection into formal design specification construction for reliable software systems," IEEE Trans. Rel., vol.65, no.1, pp.88–106, 2016.

[22] A. Hoffman, Web Application Security: Exploitation and Countermeasures for Modern Web Applications, O'Reilly Media, 2020.

[23] M. Shema, Seven Deadliest Web Application Attacks, ITPro collection, Elsevier Science, 2010.

[24] M. Biehl, Swagger and OpenAPI 2.0: Powertools for RESTful API Design, API-University Series, CreateSpace Independent Publishing Platform, 2018.

[25] D. Fett, P. Hosseyni, and R. Küsters, "An extensive formal security analysis of the OpenID financial-grade API," 2019 IEEE Symposium on Security and Privacy (SP), pp.453–471, 2019.

[26] J. Kopecký, K. Gomadam, and T. Vitvar, "hRESTS: An HTML microformat for describing RESTful web services," 2008 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology, Sydney, Australia, pp.619–625, IEEE, Dec. 2008.

[27] S.S. Alqahtani, E.E. Eghan, and J. Rilling, "Recovering semantic traceability links between APIs and security vulnerabilities: An ontological modeling approach," 2017 IEEE International Conference on Software Testing, Verification and Validation (ICST), Tokyo, Japan, pp.80–91, IEEE, March 2017.

[28] U. Klein and K.S. Namjoshi, "Formalization and automated verification of RESTful behavior," International Conference on Computer Aided Verification, vol.6806, pp.541–556, Springer, 2011.

Busalire Onesmus Emeka currently works with AlpsAlpine Co., Ltd as a System Cybersecurity Engineer. He received his Bachelors Degree from Moi University, Kenya in 2012 and his Masters and Ph.D Degrees from Hosei University, Tokyo in 2018 and 2023 respectively. His research interests include API Security, model driven secure by design approaches for API design, formal methods and formal verification approaches in software engineering.

Soichiro Hidaka is a professor of Hosei University. He received his Bachelor, Master and Doctoral degrees in Engineering from The University of Tokyo in 1994, 1996, and 1999 respectively. His research interests include programming languages, program transformations, bidirectional transformations and their application to model driven engineering. He is serving on the Steering Committee of SIG PRO of IPSJ. He is a member of IPSJ, IEICE, JSSST, ACM and IEEE.

Shaoying Liu is currently a professor of software engineering with the Hiroshima University, Japan. Previously, he was a professor with Hosei University from April 2000 to March 2020. His Research interests include formal methods and formal engineering methods for software development, specification verification and validation, specification-based program inspection, automatic specification-based testing, testing-based formal verification and intelligent software engineering environments.