Towards a Model-Driven Security Assurance of Open Source Components

Irum Rauf(✉) and Elena Troubitsyna

Åbo Akademi University, Turku, Finland
{irum.rauf,Elena.Troubitsyna}@abo.fi

Abstract. Open Source software is increasingly used in a wide spectrum of applications. While the benefits of the open source components are unquestionable now, there is a great concern over security assurance provided by such components. Often open source software is a subject of frequent updates. The updates might introduce or remove a diverse range of features and hence violate security properties of the previous releases. Obviously, a manual inspection of security would be prohibitively slow and inefficient. Therefore, there is a great demand for the techniques that would allow the developers to automate the process of security assurance in the presence of frequent releases. The problem of security assurance is especially challenging because to ensure scalability, such main open source initiatives, as OpenStack adopt RESTful architecture. This requires new security assurance techniques to cater to stateless nature of the system. In this paper, we propose a model-driven framework that would allow the designers to model the security concerns and facilitate verification and validation of them in an automated manner. It enables a regular monitoring of the security features even in the presence of frequent updates. We exemplify our approach with the Keystone component of OpenStack.

1 Introduction

The adoption of open source technology has increased tremendously in the last decade. Today most of the modern enterprises are centered around open source technology. The source code of open source software is distributed publicly and it is often developed in a collaborative manner.

The open source feature provides diverse design perspectives to the software. However, the open source software are subject to frequent updates by unknown users. This raises security concerns as the code can be used and manipulated in ways that were not initially intended by the organization.

In this work we present model-driven methodology to handle the security concerns of open-source software from design to implementation level. This work becomes more challenging when open source software are combined with REST architectural style. The adoption of REST architecture provides additional benefits of scalability and extensibility to the software encouraging providers to offer their services to a wider audience and add more features with more convenience.

A. Romanovsky and E.A. Troubitsyna (Eds.): SERENE 2017, LNCS 10479, pp. 65–80, 2017.
DOI: 10.1007/978-3-319-65948-0_5

The use of REST APIs require usage of design methodologies and security mechanisms that can handle stateless protocol for stateful applications.

Our approach to handle security concerns for REST compliant open-source software builds upon the use of *Design by Contract* strategy [18]. Contracts use preconditions and postconditions for the methods of a class to identify correctness of the program. They are capable of detecting change in the state of the program, identify when a certain piece of code violates the pre-defined conditions and can be used for fault localization. We used contracts with models to provide Security and Rest compliant UML Models (SecReUM). By using model-based test generation approach, we can generate test cases from SecReUM that can validate the behavior of the software. In addition, SecReUM can be used to provide an online/offline monitoring mechanism for KeyStone.

We exemplify our approach with the Keystone component of OpenStack. OpenStack is an open-source software platform for cloud computing that offers REST interfaces to provide IaaS (Infrastructure as a Service). The main characteristics of OpenStack include scalability, flexibility, compatibility, and openness [27]. The open source nature of OpenStack and encouragement of its partners has made it one of the most prominent cloud computing paradigm. It is deployed in various companies worldwide that have data volumes measured in petabytes and are scalable up to 60 million virtual machines and billions of stored objects [21]. Keystone offers identity service in OpenStack for authentication and authorization. This makes it a critical component of OpenStack as it serves as a gateway to all its assets.

The objective of our work is to provide an engineering solution to security experts to periodically monitor their open-source software and identify any security loopholes that may arise due to frequent updates to code in a collaborative and open environment. The use of model-driven approach facilitates an automated approach to validate the open-source components.

The paper is organized as follow: Sect. 2 briefly explains Keystone and its interface. Section 3 presents an overview of our overall approach. Section 4 presents our overall approach and Sect. 5 shows generation of contracts with security features. Section 6 presents the related work and Sect. 7 concludes the paper.

2 Keystone Open Stack

Keystone is the centralized identity service of OpenStack that offers authentication and authorization. KeyStone authenticates a user by generating a token. Token can either be scoped or unscoped depending on client's request and the configured policy of KeyStone. An unscoped token identifies a user without identifying any project scope, roles etc., whereas a scoped token provides authorization information of user for particular projects or domains. Figure 1 shows how KeyStone authentication and authorization mechanism is used with OpenStack. The client sends the authentication request to KeyStone and is sent back an *Identity Token.* This token is used by the client to request services from other OpenStack components. These services validate the identity of the client by sending the message directly to KeyStone.

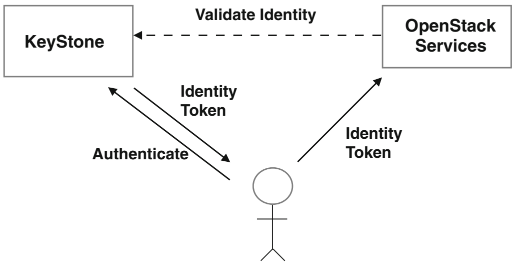

Fig. 1. KeyStone overview [4]

KeyStone offers REST API in compliance with OpenStack policy. An important feature that distinguishes REST from its contemporary SOAP-based APIs is the concept of resources. REST services expose their functionality as resources and each resource has a unique URI that provides *addressability*. CRUD (create, retrieve, update and delete) operations can be performed on resources using standard HTTP methods. These HTTP methods are considered as application-level constructs that the programs can use to interact with another program over the network in a standard manner with well-defined semantics [29]. This implies that only HTTP request methods (GET, PUT, POST, DELETE) can be invoked on KeyStone resources. In order to offer scalability, the *statelessness* feature of REST is ensured by treating every request independently. This means that every request from the client should contain all the information that is required to process it and the server is not responsible of keeping any context information with it. Each resource, when invoked via a URI and standard HTTP method, responses with response code and resource representation which contains data about resource attributes and links to other resources. The HTTP response code is a numeric code that tells the clients whether the request went successful or not. HTTP has a list of status codes that reveal how the request went [11], for example, 200 means the request was successful, 404 means the resource was not found and 403 implies that it is forbidden to make this request on this resource. The client machine interpret these response codes to know how their request went. The links in resource representation connect resources to each other and the service client gets an experience of *connectivity* between resources, i.e., moving from one resource to another.

The features of *connectivity* and *uniform interface* allows use of existing tools and infrastructure like web crawlers, curl, caches etc. The addressability requirement (specially when using hierarchical addresses) helps to provide extensibility and the statelessness requirement simplify the development of systems that can handle many service requests simultaneously facilitating scalability [25].

Listing 1.1 below shows an excerpt of POST method on *tokens* resource in KeyStone using *curl* [2] for authentication. This method is called to authenticate a user with his *name* and *password*. The payload contains JSON data that provide the required information.

```
curl -i \
  -H "Content-Type: application/json" \
  -d '
{ "auth": {
    "identity": {
      "methods": ["password"],
      "password": {
        "user": {
          "name": "admin",
          "domain": { "id": "default" },
          "password": "adminpwd"
}}}}
}' \
  http://localhost:5000/v3/auth/tokens ; echo
```

Listing 1.1. POST method for KeyStone [1]

The contemporary SOAP based services are operation centric and are based on WS-* protocol stack (SOAP, WSDL, etc.). They use different specifications built on top of each other to address different tasks. For example, WS-Resource Framework [6] and WS-Transfer [12] are commonly used to model state and WS-Security [10] is used for authentication. A common approach to invoke SOAP-based service is to call a POST method with a SOAP envelope as shown in Listing 1.2 where *curl* is used to invoke a POST method to an authentication service. All the information about the request parameters and method call are put inside the body of SOAP (*request.xml*). The server receives the request, opens the SOAP envelope and understands the message request. This means the SOAP messaging protocol is used to just transfer the messages and the semantics of the method call are determined by the message contents.

```
 curl --header "Content-Type: text/xml;charset=UTF-8" --header
      "SOAPAction:
      \"http://api.../IAuthenticationService/ClientLogin\"" \"--data
      @request.xml http://11.22.33.231:9080/AuthenticationService.svc
// Contents of request.xml
 <?xml version="1.0"encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Header>
    <Authentication xmlns="http://tempuri.org/">
      <Password>string</Password>
      <UserName>string</UserName>
    </Authentication>
  </soap:Header>
  <soap:Body>
    <HelloWorld xmlns="http://tempuri.org/"/>
  </soap:Body>
</soap:Envelope>
```

Listing 1.2. WS-Security Username Authentication [5]

Thus, the lightweight message handling mechanism and distinct features of REST architectural style make it a popular choice for adoption.

3 Overall Approach

Open-source software are open to changes and are updated frequently by different users. It becomes a challenge for in-house developers and service providers of the open-source software to validate periodically that the software continues to comply with its functional and security requirements. In a usual setting, the in-house software/security team manually look for changes and run different type of analysis techniques, ranging from manual code-inspections to running different testing tools, to identify errors. Our work provides model-driven security assurance framework for open-source software in an automatable manner. This enables the providers of open-source software to periodically verify and validate their software for the functional and security requirements it promises to deliver.

The framework is presented in Fig. 2. The framework consists of three main steps: (1) Designing (2) Generating Contracts (3) Testing. The specifications and implementation of the open source software, that are publicly available, are taken as input. The security requirements for the system are provided by security experts and also taken as an input. These three entities are marked as grey boxes in Fig. 2 to indicate their availability beforehand.

In the first step, our Security and REST compliant UML Models (SecReUM) are designed using our approach detailed in Sect. 4.

In the second step, we build upon the design by contract strategy and generate contracts from SecReUM that are implemented as code skeletons. These code skeletons are enriched with method contracts using our model-to-code transformation tool [24] and are then manually updated with security contracts and requirements, using information from SeCReUM, along with the method implementations. The code-skeletons are implemented as wrapper on top of the open-source software. A wrapper program is capable of invoking another program, perhaps with a larger body of code, by providing an interface to call. Implementation of a wrapper on top of the open-source software under test is an important component of our model-driven security assurance framework. This wrapper is maintained in-house and is updated as specifications of open source software are updated or in case of new security specifications.

The third step of our framework is *Testing* in which test cases are generated using different model-based test generation approaches from SecReUM. These test case are run against the wrapper program, generated above, to validate the implementation of open source software. Thus, by periodically running same test suites (in case the specifications are unchanged) or updated one (in case the specifications are changed), the implementation of open source software can be validated and errors can be identified using pass/fail results of the test cases.

The traceability of security requirements is also an important part of our approach. The security requirements are included as part of UML specifications and are used during validation to identify coverage level of our test cases. These requirements can be traced back to errors in the models and implementations in case of failure. This help the developers and security experts in better analysis of the system. In addition, the unfulfilled pre- and post-conditions help in localizing the faults in the implementation for both functional properties and the non-functional properties, e.g., security.

In addition to testing, the models along with implemented wrapper can also be used to provide verification of specifications and can also serve as a monitor to identify when a certain piece of updated code violates the functional or security requirements.

In this paper, we focus in detail on our designing and contract generation approach, presented in Sects. 4 and 5, respectively. The model-based test generation from SecReUM is out of scope of this paper and hence not addressed. However, for validation, we can not only benefit from our previous work for validating behavioral REST interfaces [26] but can also take advantage of large body of work done in generating test cases from behavioral contracts using UML as a familiar notation.

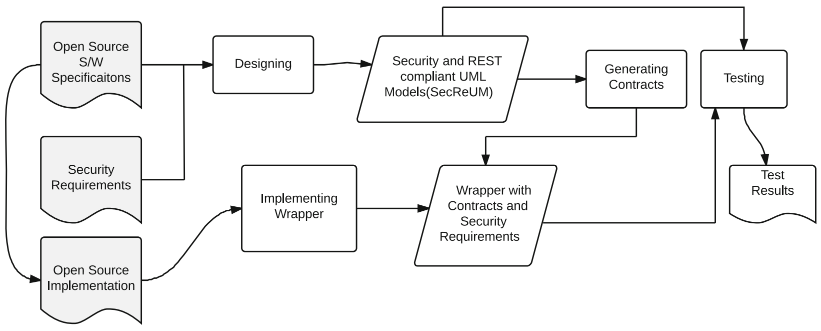
Open Source S/W Specificaitons
Security Requirements
Open Source Implementation
Designing
Security and REST compliant UML Models(SecReUM)
Generating Contracts
Testing
Test Results
Implementing Wrapper
Wrapper with Contracts and Security Requirements

Fig. 2. Model-driven framework for security assurance

4 Modeling Approach for SecReUM

REST APIs use stateless protocol but they can be used to provide applications with complex behavior having stateful behavior. The stateful services require that a certain sequence of method invocations must be followed in order to fulfill the service goals. For example, in order to delete a user in KeyStone, the user must first authenticate herself in *admin* role and get a scoped token. The benefit of giving a stateful view to this behavior of KeyStone facilitates the understanding of KeyStone behavior and helps in validating the functional and non-functional behavior of KeyStone by defining conditions under which different methods can be invoked.

The UML standard provides different types of diagrams that can model the system from different viewpoints [28]. We model the static structure and behavioral interface of a REST service with a UML class diagram and a UML state machine, respectively. Both the diagrams are defined with additional constraints to represent REST features as explained in Sects. 4.1 and 4.2. Our previous work models stateful behavior of REST services [23]. In this work, we extend our modeling approach with technique to integrate security concerns in models. Figures 3 and 4 give an example of how we model the REST interface of KeyStone. We model the behavioral interface of KeyStone from the viewpoint of our wrapper program that will invoke the KeyStone and can constrain the user to invoke the service under right conditions and service provider to fulfill the functionality expected from it.

4.1 Resource Model

The static structure of the REST service is represented with a resource model. The resource model is a class diagram that describes the resources that constitute the service and the relationships between them. The information about allowed methods on the resources is inferred from the behavioral model. All the attributes are public since they are available on public APIs. Figure 3 shows an excerpt of the resource model for KeyStone with our wrapper program. It consists of five resource namely, *SecKS*, *Token*, *Project*, *User*, *Role*. *SecKS* represents our security KeyStone wrapper which is connected to KeyStone via *Token* resource.

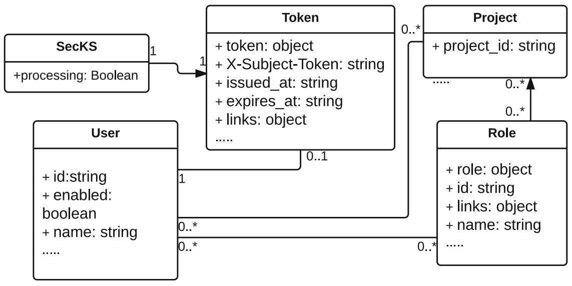

Fig. 3. Resource model for KS security wrapper (SecKS)

4.2 Behavioral Model

The purpose of the behavioral model is to describe the dynamic structure of behavioral interface of a REST service and is represented by a UML state-machine. Figure 4 shows an excerpt of behavioral interface of KeyStone and provides information on what methods a user can invoke on a resource and under what circumstances. Any client can invoke the service to request the token but only an *admin* user (shown as actor) can delete a user. If the client is valid, the token is generated, otherwise not.

A UML state-machine has transitions that are triggered by method calls and each state has a *state invariant*. State invariant is a boolean condition that is true when service is in that state and otherwise false.

In our work, we define invariant of a state as a boolean expression over addressable resources. In this way, the stateless nature of REST remains uncompromised since no hidden information about the state of the service is being kept between method calls. We have used OCL to define state invariants in behavioral models of REST services [20]. The UML specification proposes the use of OCL to define constraints in UML models, including state invariants. OCL is well supported by many modeling tools [13,14].

In Fig. 4, an OCL expression of $Token.token-> size() = 0$ in state $Token\_Not\_Granted$ means that the response for invoking GET on token resource was not 200, meaning either the resource does not exist or is not reachable to infer anything about its state. Similarly, $Token.token-> size() = 1$ implies the response for invoking GET on token resource was 200, meaning the resource exists. The state invariant $[self.processing = False\ and\ Token.token-> size() = 1]$ for $Token\_Granted$ specify that whenever a token is requested, as a result KeyStone can generate a token and it should not be processing the request (token generation is an asynchronous call). Thus, in order to define state with stateless protocol REST, we define the state invariant as a predicate over resources.

In addition, we constrain our behavioral model to have only side-effect methods, i.e., PUT, POST and DELETE methods as method calls for a transition. This is because only these HTTP methods are capable of making any changes to resources.

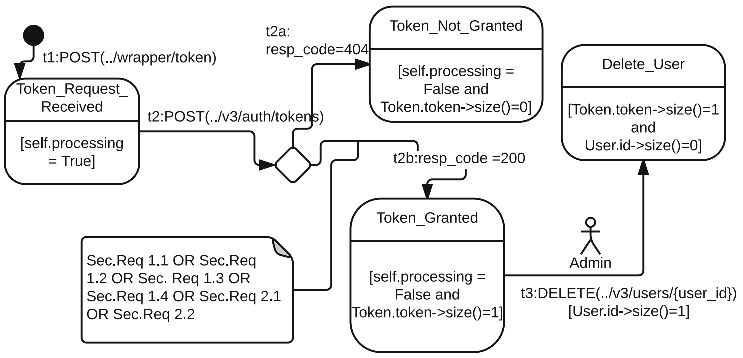
t1:POST(../wrapper/token)
Token_Request_ Received
[self.processing = True]
t2:POST(../v3/auth/tokens)
t2a: resp_code=404
Token_Not_Granted
[self.processing = False and Token.token->size()=0]
Delete_User
[Token.token->size()=1 and User.id->size()=0]
t2b:resp_code =200
Token_Granted
[self.processing = False and Token.token->size()=1]
Admin
t3:DELETE(../v3/users/{user_id}) [User.id->size()=1]
Sec.Req 1.1 OR Sec.Req 1.2 OR Sec. Req 1.3 OR Sec.Req 1.4 OR Sec.Req 2.1 OR Sec.Req 2.2

Fig. 4. Behavioral model for KS security wrapper (SecKS)

5 Generating Contracts from SecReUM

Stateful behavior of a software requires a certain order of method invocation or the conditions under which the methods can be invoked. These condition, i.e., the pre- and post-conditions of a method are called contracts. This information together with the expected effect of an operation become part of the behavioral interface of a service. Our design approach preserves the sequence of method invocations and contains behavioral information specifying the conditions under which these methods can be invoked.

5.1 Method Contract with Functional Requirements

The method contracts can be generated from the behavioral model. The precondition of a method should be true in order to fire the method in behavioral model as it defines the conditions under which a method is allowed to be invoked by the client. We say that if a method m triggers a transition t in a state machine, then the precondition for method m is true if the invariant of the source state of transition t and the guard on t is true. The post-condition constraints the implementation to provide the functionality expected from it as specified in its specification document. Thus, the post-condition states that if the precondition for invoking a method is true then its post-condition should also be true. We say, that the postcondition of method m is true if the conjunction of state invariant of target state of t and the effect on transition t are true provided its pre-condition is true. The implication principle encompasses the stateful behavior since same method can be fired from different states of the system and have different results. Thus, if the method is fired with certain pre-conditions then the corresponding post-condition for that method should be true.

The re-evaluation of the precondition of a method for evaluating the post-condition may not return the same values, i.e., before the method execution, since after the method execution values of some of the resources may change. This situation is kept safe by saving the resource values before method execution in local values in the wrapper. The values of these variables are later used to calculate the post-condition. We believe this is not computationally expensive as we do not need to save the copy of the whole resource/s but only the values that constitutes guards and invariants that are enabled. Usually, that only requires few bits of storage per method.

The method contract for method POST on *t2* can be written as under. This listing does not contain information about security requirements for invoking the method.

```
PreCondition(POST(../v3/auth/tokens)):
(self.processing = True)

PostCondition(POST(../v3/auth/tokens)):
[(self.processing = True)==>
(self.processing = False and token.token->size()=1) or
(self.processing = False and token.token->size()=0)]
```

Here, the post-condition implies that whenever a POST method is invoked on *tokens* resource from the SecKS(wrapper), SecKS is in processing state implying an asynchronous behavior. SecKS should eventually get a reply (the wrapper should not stay in processing state) and a token should either be created or not. The security requirements for generating a token and their inclusion in the contract of POST method on *tokens* are detailed in Sects. 5.2 and 5.3

A DELETE method on *User* resource will delete the user from the system and only an authorized user, i.e. an *admin*, can invoke this method. Sections 5.2 and 5.3 explain how authorization is handled in our approach. The method contract for method on *t3* can be written as under without any authorization information.

```
PreCondition(DELETE(../v3/users/{user_id})):
(self.processing = False and token.token->size()=1)

PostCondition(DELETE(../v3/users/{user_id})):
[(self.processing = False and token.token->size()=1) and
    user.id->size()=1==>
(token.token->size()=1 and user.id->size()=0]
```

For detailed description on how contracts are generated from state-machines under different scenarios, readers are referred to [22].

5.2 Security Requirements in OCL

The security requirements are usually specified by security experts. We expect these security requirements to be specified in tabular format for each method. These specifications of security requirements in a tabular format are then translated to OCL manually. These OCL-based security requirements become part of method contract during code transformation process as shown in Sect. 5.3.

The functional and security requirements for Keystone at the application level are not clearly separable. This is because the KeyStone functionality is to validate the identity of the user, his roles and access rights before generating scope or unscope token. The security requirements on KeyStone also impose the same semantics. We classify them under security requirements since the security experts expect these behaviors from KeyStone at the application level to assure its security. We explain our approach with two important security concerns, authentication and authorization. Authentication is explained with transition *t2* and authorization is explained with transition *t3*.

Authentication: Authentication is an important security concern that require that only the user with right credentials is able to enter the system. It is also considered as one of the top three security concerns addressed by existing model-driven security engineering approaches [19]. In Fig. 4, an authentication request to KeyStone triggers transition t2. The security requirements attached to t2 are listed in Table 1.

These security requirements are written in OCL. For example, the security requirement for scoped token is written as:

```
((user.credential->size()=1 or  token.token->size()=1)  and
(request.scope->size()=1 and  not  request.scope.oclIsInvalid()))
   ==> (token.token->size()=1) and token.catalog->size()=1)
```

Table 1. Requirements for authentication in KeyStone (excerpt)

No.	If	Then
1.1	User is valid and has not given scope information	An unscoped token should be generated
1.2	User is valid and has explicitly requested unscoped token	
1.3	Token is valid and has not given scope information	
1.4	Token is valid and has explicitly requested unscoped token	
2.1	User is valid and has valid scope information	A scoped token should be generated
2.2	Token is valid and has valid scope information	

In Table 1, the security requirements specify different conditions under which scoped and unscoped tokens are issued and are written in if-else format on resources and resource attributes. The security requirements can also be in a statement form enforcing some rule, for example, the authorization requirement explained in the next section.

Authorization. Authorization defines access rights of users by defining permissions on user, user roles and user groups. KeyStone determines whether a request from the user should be allowed or not based on policy rules defined in Role Based Access Control (RBAC). In Fig. 4, *t3* can only be fired by an *admin* user and not other wise. In addition, the guard value show that initially the user being deleted should exist in the system. The information of actors in the behavioral model can be realized in three ways.

(1) Developer can use this information to implement the access rights on resources and help users in understanding and writing correct authorization headers. Different authentication mechanisms can be implemented to control access to resources [3]. In case, Basic authentication mechanism is implemented, client sends the user name and password to the server in authorization header. The authentication information is in base-64 encoding. It should only be used with HTTPS, as the password can be easily captured and reused over HTTP.

In a typical setting, the authorization header is constructed by first combining *username* and *password* into a string "username:password" and then encoded in based64. A typical authorization header in Basic authentication is shown below:

```
DELETE /v3/users/22/ HTTP/1.1
Host:http://localhost:5000/v3/
Authorization: Basic aHR0cHdhdGNoOmY=
```

In case an anonymous requests for a protected resource, HTTP can enforce basic authentication by rejecting the request with a 401 (Access Denied) status code.

```
HTTP/1.1 401 Access Denied
WWW-Authenticate: Basic realm="User"
 Content-Length: 0
```

For KeyStone, authorization to resources is check with *token*. A typical call from *curl* to access *User* resource using user's *token* is given as:

```
curl -s \ -H"X-Auth-Token:
     $OS_TOKEN"\"http://localhost:5000/v3/users"
```

(2) The security requirements can be attached as predicates of boolean variables to transitions and translated to code as such. All the boolean variables for security requirements are initialized to be false, e.g. $sreq1 = False$. Whenever, the postcondition of a requirement is true in the implementation, the boolean variable is set as True, $sreq1 = True$. The boolean values of these security requirements are displayed to the user after the system is tested with different test cases. This added feature gives clear information to security experts as to what security requirements are satisfied and in identifying the met and unmet security requirements by the system without looking into the implementation details.

(3) It becomes part of method contract. The security requirement for authorization is: *Only an admin user can delete a user.* In OCL, it is written as: $user.role =$ '*admin*'.

This can be specified in UML as notes (not shown in Fig. 4 due to space limitation). In the next section, we define rules on how they becomes part of the method contract.

5.3 Method Contracts with Functional and Security Requirements

The security requirements are merged with functional requirements during the translation process to code. In our example, the KeyStone service is invoked by POST method on the token resource ($POST(../v3/auth/tokens)$). We populate our definition of contracts with security requirements given above such that:

- The statement in *if* clause become part of the method pre-condition
- The statement in *else* clause become part of the method post-condition
- The statement/s that are not part of *if-else* clause become part of both the pre- and post-condition. By checking the rule in pre-condition, the user request is validated before processing the method and causing undesired changed in the system. By placing in the post-condition, the system is validated that it behaves as expected and does not do what it is not required to do. This serves as a double check on security requirements.

We, thus, require that in order for KeyStone to generate a token the following method contract must be met:

```
PreCondition(POST(../v3/auth/tokens)):

[(self.processing = True and (user.credential->size()=1 or
token.token->size()=1)
and
((request.scope->size()=1 and request.scope <> 'unscope' and not
    request.scope.oclIsInvalid())
or (request.scope->size()=0 or request.scope.oclIsInvalid() or
request.scope = 'unscope' ))]

PostCondition(POST(../v3/auth/tokens)):
[((user.credential->size()=1 or
token.token->size()=1) and
request.scope->size()=1 and request.scope <> 'unscope' and not
    request.scope.oclIsInvalid())==>
(self.processing = False and token.token->size()=1 and
    token.catalog->size()=1)
or (self.processing = True and request.scope->size()=0 or
    request.scope.oclIsInvalid() or
request.scope = 'unscope' ) ==> (self.processing = False and
    token.token->size()=1) and token.catalog->size()=0)
]
```

The preconditions in the listing above shows the boolean expression that should be true for invoking a POST on KeyStone for either scoped or unscoped token. The postcondition circumscribes different scenarios for scoped and unscoped token. In order to return an unscoped/scoped token, the previous values, i.e. the values before method invocation, are checked. If the previous values require an unscoped/scoped token then the response of method calls are checked to ensure if unscoped/scoped token is actually delivered. The previous values, i.e., the values before the method invocation are stored as local variables in the wrapper program.

Similarly, for authorization, the method contract for DELETE on user resources is given as:

```
PreCondition(DELETE(../v3/users/{user_id}))):

[self.processing = False and token.token->size()=1 and
user.id->size()=1 and user.role='admin']

PostCondition(DELETE(../v3/users/{user_id})):
[(self.processing = False and token.token->size()=1 and
user.id->size()=1  and user.role='admin') ==>
(token.token->size()=1 and user.role='admin' and
user.id->size()=0)]
```

In this listing, *user.role* = '*admin*' is checked before invoking DELETE method on *User* resource to ensure that user with the right credentials is making the desired change in the system. Interestingly, *user.role* = '*admin*' is also a part of the post-condition, i.e., the credentials of the user are checked before and after the method execution to ensure that the system change is made by the right user. This double check of the security requirement for authorization provides added security and guards the system against malicious user during the communication.

6 Related Work

Research in using models to develop and analyze secure systems has been an active area of research for more than a decade. The work of Nguyen et al. [19] provides a comprehensive review of efforts done in the area of model-driven development of secure systems. Their work encompasses various modeling approaches like UML-based approaches, UML profiles, DSLs and aspect oriented approaches and analyzes them for their support for model-to-code and model-to-model transformations, verification, validation and different types of security concerns.

UML has been used much to model security concerns. Some approaches use only UML (e.g., [7], *MDSE@R* [9], *AOMSec* [15] etc.) and some use UML profiles(e.g., SECTET [8], *UMLsec* [16], etc.)

In [7], Abramov et al. present a model-driven approach to integrate access control policies on database development.

SECTET [8] provides a model-driven security approach for web services. They also use OCL to define constraints on UML to provide access control. The approach generates XACML policy files that provide a platform independent policy for enforcing the access control policy. The SECTET framework mainly addresses authorization and provides state-dependent permissions that are not applicable to REST interfaces. *UMLsec* [16,17] provides a comprehensive and consistently progressing approach to formally analyze the security properties. *MDSE@R* [9] provides a UML profile based approach that uses aspect-oriented programming to integrate security concerns at the runtime. *AOMSec* [15] also uses aspect-oriented approach to model security mechanism and attacks to the system. A detailed analysis of existing literature is out of scope of this paper. However, compared to previous work our work strongly relies on existing UML without the need of any new profiles. This gives the benefit of using many well-known and mature tools with a wide user base for our approach. Our work also caters well with the stateless nature of REST APIs.

7 Conclusions

Security experts are often looking out for ways to assure that their security expectations from a system are met. This becomes even more challenging in an open-source environment that encourages collaborative environment between developers that are working within a controlled environment and developers that are outside a controlled boundary. Our approach provides a security assurance framework that facilitates the security experts by providing a semi-automatable approach for validating the system under study for its behavior. We show how the security concerns can be integrated into the behavioral models of REST services and how method contracts can be generated from them that can be used to validate any security loopholes in the open source software in case of frequent updates. We address authentication and authorization of open source software using models and provide series of steps on how the security requirement can be combined with functional contracts. The approach is applied on the KeyStone component of OpenStack. In our future work, we plan to provide automation of security concerns to code and extend our work with other security concerns.

References

1. API Examples using Curl. https://docs.openstack.org/developer/keystone/devref/api_curl_examples.html. Accessed June 2017
2. cURL. http://curl.haxx.se/. Accessed 20 May 2017
3. HTTP Authentication. http://www.httpwatch.com/httpgallery/authentication/. Accessed 20 Aug 2013
4. KeyStone Security and Architecture Review. https://www.openstack.org/summit/openstack-summit-atlanta-2014/session-videos/presentation/keystone-security-and-architecture-review. Accessed June 2017
5. SOAP Request and CURL. http://dasunhegoda.com/make-soap-request-command-line-curl/596/. Accessed June 2017
6. Web services resources framework (wsrf 1.2). https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsrf. Accessed 01 Nov 2013
7. Abramov, J., Anson, O., Dahan, M., Shoval, P., Sturm, A.: A methodology for integrating access control policies within database development. Comput. Secur. **31**(3), 299–314 (2012)
8. Alam, M.M., Breu, R., Breu, M.: Model driven security for web services (MDS4WS). In: Proceedings of INMIC 2004 - 8th International Multitopic Conference, pp. 498–505. IEEE (2004)
9. Almorsy, M., Grundy, J., Ibrahim, A.S.: Adaptable, model-driven security engineering for SaaS cloud-based applications. Autom. Softw. Eng. **21**(2), 187–224 (2014)
10. Atkinson, B., Della-Libera, G., Hada, S., Hondo, M., Hallam-Baker, P., Klein, J., LaMacchia, B., Leach, P., Manferdelli, J., Maruyama, H., et al.: Web services security (WS-Security). Specification, Microsoft Corporation (2002)
11. Berners-Lee, T., Fielding, R., Frystyk, H.: Hypertext transfer protocol-HTTP/1.0 (1996)
12. Davis, D., Malhotra, A., Warr, O.K., Chou, W.: Web services transfer (WS-Transfer). World Wide Web Consortium, Recommendation REC-ws-transfer-20111213 (2011)
13. Demuth, B., Wilke, C.: Model and object verification by using Dresden OCL. In: Proceedings of the Russian-German Workshop Innovation Information Technologies: Theory and Practice, pp. 81–89 (2009)
14. Garcia, M., Shidqie, A.J.: OCL compiler for EMF. In: Eclipse Modeling Symposium at Eclipse Summit Europe (2007)
15. Georg, G., Ray, I., Anastasakis, K., Bordbar, B., Toahchoodee, M., Houmb, S.H.: An aspect-oriented methodology for designing secure applications. Inf. Softw. Technol. **51**(5), 846–864 (2009)
16. Jürjens, J.: Towards development of secure systems using UMLsec. In: Hussmann, H. (ed.) FASE 2001. LNCS, vol. 2029, pp. 187–200. Springer, Heidelberg (2001). doi:10.1007/3-540-45314-8_14
17. Jürjens, J., Shabalin, P.: Tools for secure systems development with UML. Int. J. Softw. Tools Technol. Transf. **9**(5–6), 527–544 (2007)
18. Meyer, B.: Applying ‘design by contract’. Computer **25**(10), 40–51 (1992)

19. Nguyen, H.P., Kramer, M., Klein, J., Traon, Y.L.: An extensive systematic review on the model-driven development of secure systems. Inf. Softw. Technol. **68**, 62–81 (2015)
20. OMG: OCL, OMG Available Specification, Version 2.0 (2006)
21. Pepple, K.: Deploying OpenStack. O'Reilly Media Inc., Sebastopol (2011)
22. Porres, I., Rauf, I.: From nondeterministic UML protocol statemachines to class contracts. In: 2010 Third International Conference on Software Testing, Verification and Validation (ICST), pp. 107–116. IEEE (2010)
23. Porres, I., Rauf, I.: Modeling behavioral restful web service interfaces in UML. In: Proceedings of the 2011 ACM Symposium on Applied Computing, pp. 1598–1605. ACM (2011)
24. Rauf, I., Porres, I.: REST: from research to practice. In: Wilde, E., Pautasso, C. (eds.) Beyond CRUD, vol. 2029, pp. 117–135. Springer, New York (2011). doi:10.1007/978-1-4419-8303-9_5
25. Rauf, I., Ruokonen, A., Systa, T., Porres, I.: Modeling a composite restful web service with UML. In: Proceedings of the Fourth European Conference on Software Architecture: Companion Volume, pp. 253–260. ACM (2010)
26. Rauf, I., Siavashi, F., Truscan, D., Porres, I.: Scenario-based design and validation of REST web service compositions. In: Monfort, V., Krempels, K.-H. (eds.) WEBIST 2014. LNBIP, vol. 226, pp. 145–160. Springer, Cham (2015). doi:10.1007/978-3-319-27030-2_10
27. Sefraoui, O., Aissaoui, M., Eleuldj, M.: Openstack: toward an open-source solution for cloud computing. Int. J. Comput. Appl. **55**(3) (2012)
28. OMG Uml. 2.0 superstructure specification. OMG, Needham (2004)
29. Webber, J., Parastatidis, S., Robinson, I.: REST in Practice: Hypermedia and Systems Architecture. O'Reilly Media Inc., Sebastopol (2010)