

# Finding Logical Vulnerability in Policies Using Three-Level Semantic Framework

Karolina Bataityte<sup>1</sup>, Vassil Vassilev<sup>2</sup>, and Olivia Jo Gill<sup>1</sup>

<sup>1</sup> School of Computing, London Metropolitan University, London, UK

<sup>2</sup> Cyber Security Research Centre, London Metropolitan University, London, UK  
kab0863@my.londonmet.ac.uk, {v.vassilev,o.gill}@londonmet.ac.uk

**Abstract.** We present the continuation of our work on a three-level framework, which can be used to model and analyze the identification-authentication- authorization policies. Finding the gaps in such policies is challenging. We explore the cases when operations become accessible to the user because of flawed or missing authentication methods. Our objective is to model the domain and find such vulnerabilities. Our proposed framework has three levels. Each level is built on top of a previous one. The first is ontological, where we model the static domain in OWL; the second is logical, where we model the dynamic using SWRL; and the third is analytical level, where we utilize the reasoner to get the results. In this paper, we present the algorithm, which finds vulnerable situations in the policies or confirms that there are no vulnerable situations. We have modelled a couple of policies from different user-based applications to validate our approach as well as demonstrate the feasibility of using it on policies from the actual systems.

**Keywords:** Logical Vulnerability · identification, authentication and authorization policies · OWL · SWRL · reasoner

## 1 Introduction

The past decade has seen a rapid increase in the usage of online accounts in various fields, including banking and shopping. The account security concerns must be addressed to ensure that sensitive information stored in these accounts and operations are secure. One crucial aspect of maintaining account security is using identification, authentication, and authorisation policies in the applications. Identification refers to information that a user knows, has, or is, while authorisation is the method used by the system to verify the identity of the user, and authorisation determines the permissions to perform certain operations for an authorised user. Users must always go through this process to log in, and once logged in, the user is authorised to do certain operations. After a user is logged in, additional procedures might exist to get permission for other operations or information. We are going to call such procedures **additional authentication** e.g., once a user is logged in, he needs to enter a one-time generated code in order to be able to update a password. However, some fraudulent users attempt

to exploit the applications by manipulating available information and operations to identify gaps in these policies and perform operations that lack proper authorisation without needing to use additional technical tools. This way, the fraudulent user can gain information and access to operations, potentially resulting in theft or damage. In-depth knowledge of policies is usually required to find and address these gaps in the policies. Unfortunately, it can be challenging to identify a gap or workaround in the policies rules of the application if one is unsure what exactly to look for. Typically, these gaps are only addressed after the application has already been exploited. These gaps in the policies can also be seen as logical gaps in access control, **which legitimately permits something which should not be permitted**.

We propose a three-level framework to model these identification - authentication - authorisation policies. We can tell what information and operations a user (legit or not) can gain by having a starting point. Our approach utilises semantic technologies. We use ontologies to model policy dynamics, which can also be challenging; however, it allows us to use reasoning for the analysis. This work is based on a previously published paper where we introduced the first two levels of the framework for modelling the domain and policies in depth [2]. This paper focuses on the analysis part to find vulnerabilities using the framework.

This paper has been divided into six sections. The first section introduces the topic and proposed framework. The second section expands on the problem statement and outlines some related work. The third section summarises the first two levels of the framework. The fourth section presents the algorithm on the third level in great detail. The fifth section presents some validation and evaluation. Finally, the sixth section concludes our work.

## 2 Background

In this section, we would like to delve deeper into the problem statement and present some literature review to better understand the context and existing research related to the issues at hand.

### 2.1 Deeper Understanding of Problem Statement

Our problem statement originated from an industry scenario where users were granted the ability to update their phone numbers through a mobile application, leading to the growth of vulnerabilities. They noticed an increase in fraudulent transactions. However, the fraudulent transactions decreased after implementing additional authentication measures during the phone number update operation. Out of this, we decided to investigate the identification - authentication - authorization policies for the operations and observe what information the system gives and what operations can become accessible. Can a user pretending to be a true user access operations that should not be accessible?

We would like to illustrate this with an example. A true user possesses complete knowledge and is granted access to all operations, while a fake user has

limited information and restricted access. Let us consider the following set of operations with their policies:

1. Login requires to provide a password.
2. To update an email address requires input only a new email address.
3. To add a new recipient requires an input of a security code, which the user receives in their email address.
4. To send money requires input (choose) a recipient and an amount of money to send.

Although this restricts access, a fake user with partial information may gain access to operations they should not have. In the policies above, we see that adding a new recipient requires inputting an additional security code, which is sent to the user's email address. This shows that this operation has an additional authentication step, which should add additional protection. However, let us consider the following:

- A fake user obtained a password and logged in to a system pretending to be someone else (we say that a password here is known information to this user).
- A fake user would like to add a recipient to send some money. This is how the money could be stolen from the account. However, it cannot be done since it requires a code sent to the email address stored in the account.
- A fake user can update an email address by providing his/her email address (we say that an email address here is gained information to this user).
- Now, a fake user can add a new recipient since the code is sent to the new email address.
- Money is stolen.

In this example, we can clearly see how enabling to update an email address without any additional authentication opened a vulnerability to add a new recipient without an **effective additional authentication**. Therefore, we would say that the operation and policy to add a new recipient has a vulnerability. This can be fixed by either changing the method of additional authentication when adding a new recipient or by adding an additional authentication when updating an email address.

This analysis technique is beneficial for verifying that the current policies do not have any gaps, meaning that the user can complete operations with known parameters and cannot gain information, which would allow completing more operations. **We define a gap in a set of policies if a user can obtain information that would enable them to perform additional actions beyond what is currently permitted. We say that this is a logical vulnerability** because this fake user does not need any technical tools; it is enough to manipulate the system's policies to find a gap. If the user is legit, she/he knows everything, so there is nothing to obtain, and everything is legitimately accessible. However, the policies should prevent fake users from accessing things that require additional authentication. Of course, it is up to the policymaker to decide which operations are more important and valuable to have this additional authentication to prevent potential hackers from accessing it.

## 2.2 Security Ontologies and Access Control Models

This is a security problem; therefore, we looked at several ontologies that model security. The ontology can be used as a vocabulary to agree on terms and definitions. It also can be used to search and ask questions. Some of them model the vulnerabilities of security from the perspective of threats [10], [4], [3], [1] they also show a weakness of assets, weaknesses in policies. Some model the business process and logical vulnerabilities [6], [7], [9]. These ontologies have their methodology and taxonomy. Our work is different from theirs as we model the actual system's policies instead of having a generalised security and vulnerability model. Then, we have an algorithm to find the vulnerable policies in the modelled system.

Our problem statement also sounds a lot like evaluating the access control policies, so we reviewed some of them. There is a wide variety of access control models to manage permissions to the resources [8]. For instance, "Access Control Lists (ACLs) is a list of permissions, which are granted to a user". Mandatory Access Control (MAC) uses security labels for security clearance, and changing them is impossible. "Trust-Based Access Control" (TrustBAC) monitors the behaviour of the user, and the trust level can be reduced if the user does not behave as expected [8]. There are many more models, and some of them are more complicated than others.

We want to talk about access control from the view of identification - authentication - authorisation policies in a specific user-based system or application, which makes operations accessible to a user. We are not defining the policies or model for the policies; they are already defined in the applications. We are exploring the policies to see if they work as expected. **Policies for us are rules that tell us the needed identification, authentication, and authorisation for an operation.**

## 3 Introducing The Three-Level Framework

To formalize the third level and facilitate practical application, we must introduce and discuss various concepts and elements of the framework. We have developed and presented the three-level framework to address this problem statement [2]. Here we are going to briefly re-cap the first two levels of the framework for needed analysis on the third level.. The first level models the domain of policies and user operations in an application using Web Ontology Language (OWL) [11]. The OWL is a standardized language for expressing ontologies. We are mainly using OWL Classes, OWL Object Properties, OWL Individuals (which we will collectively refer to as classes, object properties, and individuals) and some other axioms. Our framework has defined a particular terminology presented in a table 1 below. It is important to note that object properties have domain and range restrictions. Action object property must have mandatory situation classes as domain and range (modelled in OWL), while present-at object property must have an entity class as domain and situation class as range restrictions (modelled in SWRL).

**Table 1.** Framework Terms

Framework OWL Term	Description	Examples
Situations Class ( $S$ )	These classes represent the situation in the policy rule	$S\_Login$ , $S\_SendMoney$
Entities Class ( $En$ )	These classes represent information required for the policy model	$password$ , $amount$
Actions Object Property ( $A$ )	These object properties are transitions between the situations	$A\_LoginWithPasscode$ , $A\_AddRecipient$
Present-at Object Property ( $IsInS$ )	These object properties relate entity classes with situation classes	$IsInS\_Login$ $IsInS\_SendMoney$

The second level models the dynamics using the domain model from the first level. It uses Semantic Web Rule Language (SWRL) [5], which is designed to work with OWL and extends OWL expressivity by allowing it to formulate additional rules. The SWRL rule has the following format:  $body \rightarrow head$ , where the body and the head are made up of conjunctions of SWRL atoms, which can be seen as classes and object properties. If the body is true, then the head must be true [5]. We define the following template of the SWRL rule to express the dynamics of policies and call it Action Rule; see definition 1. We also define the input and output parameters as in definitions 2 and 3.

**Definition 1.** *A template for Action Rule  $R(A)$  is an SWRL rule which describes an action, a transition from one situation to another situation along with conditions, expressed as input parameters and effects expressed as output parameters as in the template below:*

$$S\_a(?sa) \wedge A\_n(?sa, ?sb) \wedge \langle input\ par. \rangle \rightarrow S\_b(?sb) \wedge \langle output\ par. \rangle \quad (1)$$

where input and output parameters are of the form  $En\_e(?e) \wedge IsInS(?e, ?sx)$  The action rule must have only one situation, an action coming from the situation and optional input parameters in the body; the head contains one leading situation from the action and optional output parameters. When input parameters are empty, no output parameters are allowed. If input parameters are not empty, output parameters may be either empty or not.

- If  $A_{in} = \{\}$  then  $A_{out} = \{\}$
- If  $A_{in} \neq \{\}$  then  $A_{out} = \{\}$  or  $A_{out} \neq \{\}$

The rule describes the specific execution. Each rule is unique; therefore, each action is unique.

**Definition 2.** *Input parameters of action  $A_{in}$  is a set of entities such as  $En(?x)$ , which are in the body of an Action Rule  $R(A)$ . Minimum set of  $A_{in}$  is null and maximum set of  $A_{in}$  is a finite set of entities*

**Definition 3.** *Output parameters of action  $A_{out}$  is a set of entities such as  $En(?x)$ , which are in the head of an Action Rule  $R(A)$ . A minimum set of  $A_{out}$  is null, and a maximum set of  $A_{out}$  is a finite set of entities.*

One of the features of this framework is that the rules are chained via the situations, which are linked via actions. While we have transformed this into a graph, such a subject is beyond the scope of this paper. A separate paper has been submitted specifically addressing the conversion into graph structures.

Finally, the third level includes an analysis to address the problem statement. The following terms are necessary to describe the proposed algorithm. Since OWL is based on Description Logics (*DL*) [11], we will use its notations throughout the paper. *DL* is a formal knowledge representation language. The Knowledge Base (*KB*) expressed in *DL* consists of two main components: Terminological Box (*TBox*) and Assertional Box (ABox). TBox can have concept inclusion  $C_1 \sqsubseteq C_2$  and role inclusions  $R_1 \sqsubseteq R_2$ , where  $C_1$  and  $C_2$  are concepts (= classes) and  $R_1$  and  $R_2$  are roles (= object properties). ABox can have assertions of the two following forms:  $C(a)$  called concept assertions and  $R(a, b)$  called role assertion.

## 4 Finding the Vulnerabilities: An Algorithmic Approach

In this section, we will introduce our proposed algorithm to find logical vulnerabilities in access policies in detail. We want to observe what information users can collect and what operations can become available. We focus on the accessibility of operations (situations, actions) and information (parameters expressed as entities) users know and can discover. Consequently, we are going to categorize the parameters (entities) based on the following classification:

1. *Known parameters* are entities which the user knows (has or is). These parameters are mostly used as input parameters (e.g., *password*, *emailAddress*).
2. *New parameters* are entities which the user can create/update. A user does not necessarily need to know the parameter's current value to update or create them (depending on a policy). These parameters are mostly used as input parameters. These parameters have the prefix "new" in the model (e.g., the user can update the address using *newAddress* input parameter).
3. *Gained parameters* are parameters which are stored in the system, and the user does not necessarily know them at the beginning. Users can acquire knowledge of this stored information. These parameters can be stored as input or output parameters (e.g., *balance*).

Another essential concept is a special entity called *User*. It does not belong to any of the below categories. All the action rules have a user as the input and output parameters. This way, once the rule is triggered, the user-situation object property assertion is added (e.g.,  $IsInS\_UpdatePassword(user_1, S\_UpdatePassword_1)$ ). This allows us to see user-accessed situations in our simulation. In other words, it indicates that the situation can be reached. We can formally define the user-accessible situation in the following definition 4. The example of the action rule with the user follows:  $S_1(?x) \wedge User(?u) \wedge IsInS_1(?u, ?x) \wedge A_1(?x, ?y) \rightarrow S_2(?y) \wedge IsInS_2(?u, ?y)$ .

**Definition 4.** A user-accessible situation is a situation  $S$ , which has the assertion of the following form in the ABox:  $IsInS\_1(user_1, s_1)$  where  $user \sqsubseteq$  entities,  $S_1 \sqsubseteq$  situations,  $IsInS\_1 \sqsubseteq$  Preset – At,  $user(user_1)$ ,  $S_1(s_1)$ .

Our goal is to observe the accessibility in the ABox by utilising the reasoner to see inferences on individuals. We need to see inferred information to interpret the policies' flow. Reasoning is essential in our algorithm. The inference derived from the SWRL rules is applied to the individuals, which usually represent data. We direct our attention to policy rules and how different components and elements impact each other within rules; therefore, data itself does not have significance for us. At the same time, our framework terminology is unique; no existing data set could align with our required terminology. Therefore, we generate some individuals at the beginning of the algorithm. Rules are obtained differently; we can obtain the actual rules through interaction with an application, getting the necessary elements for modelling the domain and rules.

Having all of the above, we can present the algorithm 1. The algorithm 1 takes the Ontology  $O$  consisting of a fully modelled  $TBox$  taxonomy,  $ABox$  assertions and  $R(A)$  SWRL Action Rules as input for the algorithm. Also, we decide on a set of known parameters  $P$  and the set of accessible situations, which currently can have one start situation  $s_1$ . A good example would be  $AS = \{S\_Login\}$ ,  $P = \{username, password\}$ . Also, the assertions for  $P$  parameters are added to all needed situations. Lastly, we have a special entity,  $user$ , with its assertion to the start situation. We do not input the end situation as the algorithm, aka reasoning, will explore everything it can reach and stop once nothing is left. This algorithm produces a set  $P'$  of all parameters (entities) which are used by inference, meaning that it has an assertion. It also provides the set  $AS'$  of accessible situations as per definition 4, and lastly, it produces a set of vulnerable situations  $VS$ .

The algorithm starts by executing *addAssertionsForSituationsAndActions* ( $TBox$ ), which generates one unique individual assertion for each situation class (see 2 below) and action object property assertions according to the domain and range restrictions (see 3 below). This creates the basis ABox (line 1)

$$\textit{For every } S \textit{ in Situations, it is necessary that } S(s_1) \quad (2)$$

$$\textit{For every } A \textit{ in Actions, it is necessary that } A(s_1, s_2) \quad (3)$$

Then it executes *addInputParametersStartingWithPrefixNew*( $TBox$ ), which adds one unique individual assertion for entities which belong to the parameters category "New". We also add object property assertions with the relevant *present – at* object property, "new" *entity* individual and *situation* individual according to the bodies of the rules (see 4 below) (line 2).

$$\textit{For every newEn in Entities, it is necessary that newEn(newEn}_1) \quad (4)$$

For example, if body of  $R(A_1) \equiv S(?x) \wedge newEn(?q) \wedge IsInS(?q, ?x) \wedge \dots \wedge A_1(?x, ?y)$  then it is necessary that  $IsInS(newEn_1, S_1)$

---

**Algorithm 1** Get Accessed Situations, Parameters and Find Vulnerable Situations

---

**input:** OWL  $O = (TBox, ABox, R(A))$ ,  $ABox = \{User(user_1)\}$   
 set of known parameters to start  $P = \{p_1, \dots, p_m\}$ ,  
 set of accessible situations to start  $AS = \{s_1\}$   
**output:** set of known and gained parameters  $P'$ ,  
 set of accessible situations  $AS'$ ,  
 set of vulnerable situations  $VS$

- 1:  $ABox' \leftarrow ABox \cup \text{addAssertionsForSituationsAndActions}(TBox)$
- 2:  $ABox'' \leftarrow ABox' \cup \text{addInputParametersStartingWithPrefixNew}(TBox)$
- 3: boolean  $inference = true$ ,  $counter \leftarrow 0$
- 4: **while**  $Inference = true$  **do**
- 5:      $counter \leftarrow counter + 1$
- 6:      $Inference \leftarrow \text{runReasonerCheckInference}(O)$
- 7:      $P' \leftarrow P \cup \text{addEntityFromClassAssertions}()$
- 8:      $LastSetAS \leftarrow null$
- 9:     **for** each  $assertion \in ABoxes$  **do**
- 10:         **if**  $present - at(user_1, situation) \in assertion$  **then**
- 11:              $AS' \leftarrow AS \cup situation$
- 12:              $LastSetAS \leftarrow AS'$
- 13:             **if**  $counter = 1$  **then**
- 14:                  $FirstSetAS \leftarrow AS'$
- 15:             **end if**
- 16:         **end if**
- 17:     **end for**
- 18:     **for**  $p \in P$  **do**
- 19:         **for**  $r \in R(A)$  **do**
- 20:             **if**  $p \in r.getBody$  **then**
- 21:                  $situation \leftarrow \text{find situation individual for situation class in } r.getBody$
- 22:                  $entity \leftarrow \text{find entity individual for entity class}$
- 23:                  $present \leftarrow \text{get present-at object property in } r.getBody$
- 24:                  $ABox''' \leftarrow ABox'' \cup present - at(entity, situation)$
- 25:             **end if**
- 26:         **end for**
- 27:     **end for**
- 28: **end while**
- 29:  $VS \leftarrow LastSetAS.removeAll(FirstSetAS)$

---

Next, we create a boolean variable *Inference* (*true* or *false*), which is going to tell when there are no more new inferred assertions, and we set it to *true* to trigger the first inference. We also create a counter to track how many times this loop will be executed (line 3). Then, we enter a while loop, which stops once the inference does not add any new assertions (line 4). We start the loop by increasing the *counter* by one (line 5). Following that, we run the reasoner to get the inferred assertions. We do that by executing *runReasonerCheckInference(O)*, which creates the set of current assertions, executes the openllet reasoner, creates another set of assertions after running the reasoner and compares these two sets. If there are new assertions, then it returns *true*; if there are no new assertions, it returns *false* (line 6). Then, we get all entity classes that have an assertion and add them to the known and gain parameters set *P* (line 7). Note that parameters in the category "new" can become "gained". We create a new empty set *LastSetAS*, where we will save the last set of accessed situations. Therefore, the following part loops through assertions in *ABox* (line 9) and checks if there is an object assertion such as *present – at(user, situation)*, which indicates that the reasoner added this assertion and, therefore, the *situation* is accessible by the user as per definition 4 and the *situation* is added to the accessed situations list *AS* (line 10-11). Then we save the *AS'* to the *LastSetAS* in case this is the last loop (line 12), and if not, then *LastSetAS* will get set to null (line 8). Similarly, we check if this is the first loop (line 13), and if that is the case, then we save the set of accessed situations to the set *FirstSetAS*.

The following part requires some additional clarification; therefore, we are going to describe the next part with greater elaboration. We want to take all assertions that were added as a result of rules (as output parameters) and add those assertions where they are required as input parameters. Let us illustrate this with an example below.

*Example 1.* We have the following two rules *R(A1)* and *R(A5)* respectively:

$$S1(?x) \wedge newEn1(?q) \wedge IsInS1(?q, ?x) \wedge A1(?x, ?y) \rightarrow S2(?y) \wedge En3(?q) \wedge IsInS3(?q, ?y) \quad (5)$$

$$S4(?x) \wedge En3(?q) \wedge IsInS4(?q, ?a) \wedge A5(?x, ?y) \rightarrow S5(?y) \wedge IsInS5(?q, ?y) \quad (6)$$

Before executing the reasoner, we have the following *ABox*:

$$\{S1(s_1), S2(s_2), S4(s_4), S5(s_5), A1(s_1, s_2), A5(s_4, s_5), newEn1(newEn1_1), IsInS1(newEn1_1, s_1)\} \quad (7)$$

as per the first part of the algorithm. Once we run the reasoner, the head of *R(A1)* becomes true and therefore  $\{En3(newEn1_1), IsInS2(newEn1_1, s_2)\}$  assertions are added to the *ABox*. Since this parameter *En3(newEn1\_1)* is now accessible, we want to add this accessible entity as an assertion for the body of a rule so it can become true in the next inference. Therefore, we add the following to the *ABox*: *IsInS4(newEn1\_1, s\_4)* as per *R(A5)* body. Now we can run the reasoner again and the head of *R(A5)* becomes true.

In real life example, *newEn1* can be *newPassword*, which becomes *En3 = password* and then in  $R(A5)$ , the password is used to access something else, e.g., situation  $S\_AddRecipient$  in banking environment.

That is the process when the output parameters (entities) become the input parameters (new or inferred) for needed rules. Going back to the algorithm 1, we loop through the entities in the set  $P$  (line 18), and we loop through the set of rules  $R(A)$  (line 19). Then we check if the entity belongs to the body of the rule (line 20), and if it does, then we need to get the following: 1) the situation class from the rule and find the individual for it (line 21), 2) find the individual for the entity class (line 22) 3) get *present-at* object property from the body of the rule (line 23). Then, finally, add an assertion using these three elements to the *ABox* (line 23). This way, we add all output parameters (obtained inference from rules) to be input parameters where needed. The main while loop runs while there are no more inferred assertions from the reasoner (lines 4-27). By this point, we have the final sets of  $P'$  and  $AS'$ . At last, we compile the set of vulnerable situations by removing all situations in *FirstSetAS* from *LastSetAS* (line 28). We could also say that all situations in *FirstSetAS* are not vulnerable.

The first set, *FirstSetAS*, initially contains everything accessible with the initially provided parameters. Subsequently, we obtain information by inferences. We utilize this acquired information (parameters and situations) to introduce more inferred assertions. Consequently, additional authentication is derived from the gained information (parameters), rendering it ineffective in fulfilling its intended purpose of not allowing it to gain more accessibility or information. Therefore, the *LastSetAS* has all accessed situations using all known and gained parameters. Subtracting *FirstSetAS* from *LastSetAS* leaves us with the set of situations, which got accessed using the gained parameters and, therefore, are vulnerable.

#### 4.1 Process Example via a Use Case

To show the feasibility of the framework and illustrate the algorithm, we present the use case from the very start of modelling to the algorithm's results. For this use case, we will use the example described in the section 2.1, which illustrates a fragment of operations along the policies with a vulnerability. Let us start with the ontological level, which is modelled in the Protégé<sup>3</sup>. Figure 1 displays the classes and object properties. On top of classes and object properties, we have the following axiom  $codeFromEmail \sqsubseteq emailHasUser.User$ . With this, we want to say that if a user has an email address, then they can receive the identification code, so they have *codeFromEmail*. Then, we have the action rules on the logical level also modelled in Protégé (figure 1). Now we can move to the analytical level - in this case, the algorithm presented in the section 4. We will go through the main steps of the algorithm.

<sup>3</sup> <https://protege.stanford.edu/>

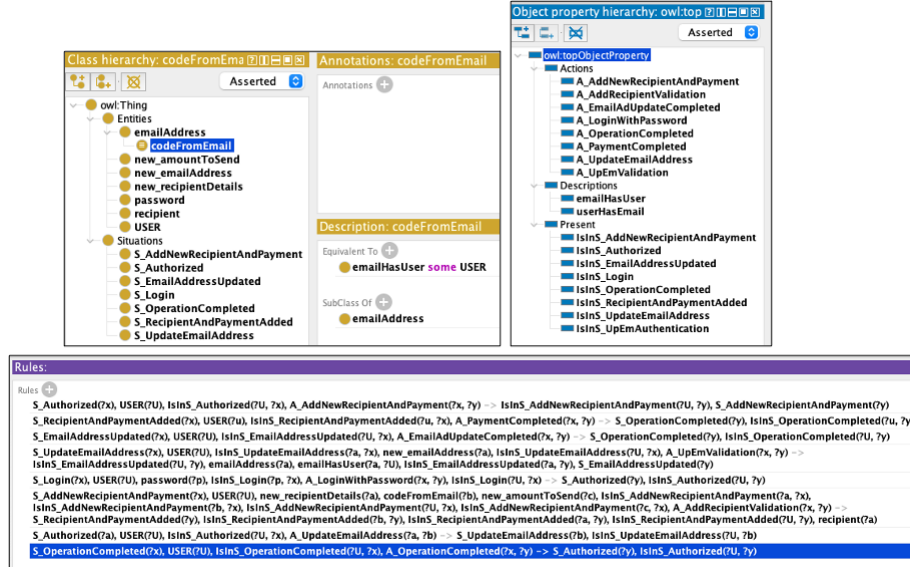


Fig. 1. Ontological & Logical Levels: Classes, Object Properties and Action Rules

1. The input for the algorithm is the ontology presented in the figures 1 and user individual. We decided the set of known parameters  $P = \{password\}$  and the set of accessible situations  $AS = \{S\_Login\}$ .
2.  $addAssertionsForSituationsAndActions(TBox)$  adds the assertions for all situations and actions to the ABox  $A$ . To save space, we show only a few:  $A = \{S\_Login(S\_Login01), S\_Authorized(S\_Authorized01), A\_LoginWithPassword(S\_Login01, S\_Authorized01), \dots\}$
3.  $addInputParametersStartingWithPrefixNew(TBox)$  adds the the assertions to the ABox  $A$  for the set  $P$  and entities in a category "new". We show three out of five here:  $A' = A \cup \{new\_emailAdd(new\_emailAdd01), IsInS\_UpdateEmailAdd(new\_emailAdd01, S\_UpdateEmailAdd), user(user01), password(password01), IsInS\_Login(user01, S\_Login01), IsInS\_Login(password01, S\_Login01)\}$
4. We enter the loop and run the reasoner for the first time. It adds all needed inference assertions from the reasoner and we get two sets, where  $AS$  is also saved as  $FirstSetAS$ .

$$P = \{password, new\_recipientDetails, new\_emailAddress, new\_amountToSend, emailAddress, codeFromEmail\}$$

$$AS = \{S\_Login\_01, S\_Authorized\_01, S\_UpdateEmailAdd\_01, S\_EmailAddUpdated\_01, S\_AddNewRecipientAndPayment\_01, S\_OperationCompleted\_01\}$$

Note: the entity  $codeFromEmail$  from the Rule  $R(A\_UpEmValidation)$  and axiom  $codeFromEmail \sqsubseteq emailHasUser.User$  becomes input param-

eter for

$R(A\_AddRecipientValidation)$ .

5. The second iteration runs the reasoner again and we get the following sets:  $P' = P \cup \{recipient\}$ ,  $AS' = AS \cup \{S\_RecipientAndPaymentAdded\_01\}$
6. the third iteration runs the reasoner and there are no more inferred assertions therefore the loop stops.
7. The last step is to compare  $FirstSetAS$  and  $LastSetAS$  where we get a set of vulnerable situations such as  $VS = \{S\_RecipientAndPaymentAdded\_01\}$

## 5 Validation and Evaluation

For the validation, we have the implementation of this framework as a proof of concept. We model the domain using classes, object properties, other axioms and SWRL rules in Protégé. We have implemented our algorithm in JAVA using OWL API and openlet reasoner. When we generate individuals, we create names for them by appending a number to a class name. For example, if the class is named "S\_Login," the corresponding individual would be named "S\_Login01".

In order to validate and evaluate our algorithm, we modelled the fragments of policies of three systems. The first is our created test system, of which the fragment is presented in the section 4.1; then, we modelled Lloyds and Revolut banking apps. The static and dynamic aspects of the latter two models were constructed by being a user of these apps, simply trying to complete operations, and observing what information the app is requesting, what information we are getting, and what operations are available. See the summary in the table 2:

We counted operations as each operation has its own set of policies. If there are two ways of performing the operation, we modelled both and counted as two (e.g., we can log in with a password or fingerprint). We modelled operations such as login, updating passwords, adding new recipients, sending money, etc. We did not count classes "*situations*" and "*entities*" as well as object properties "*actions*" and "*present – in*". The count of axioms is taken from the ontology metrics. Please note that the test system has two additional object properties, which were used for the axioms.

We hoped that by modelling the actual application, we could evaluate and validate our implementation better. We wanted to see the capability and practicability of using our framework for policies of real applications. Of course, the actual banks (the last two models) do not have any policy gaps, which we should have foreseen. We plan to tweak the models and see how changes in the policies can affect the vulnerability.

We gathered some advantages and disadvantages of our framework to evaluate its overall effectiveness, providing a comprehensive understanding of its strengths and weaknesses. During the process of modelling, we noticed that our approach can model and validate different applications using the same structure and terminology. This opens an opportunity to have the same terms for multiple applications, making it easy to compare them fairly. One of the difficulties was the fact that the rules cannot add individuals if they do not exist. We overcame

**Table 2.** The Summary of Tested Models

Counts of:	Test System	Lloyds Banking App	Revolut Banking App
Operations	4	11	8
Total Classes	24	41	37
Situations Classes	14	27	24
Entities Classes	10	14	13
Total Object Properties	38	62	49
Actions Object Properties	17	34	27
Present-at Object Properties	19	28	22
Rules	16	33	23
Object Property domain and range axioms	32	68	52
Individulas (before / after)	0 / 20	0 / 35	0 / 30
Axioms (before / after)	177 / 337	312 / 512	253 / 451
Classes assertions (before / after)	0 / 44	0 / 70	0 / 63
Object pro. assertions (before / after)	0 / 96	0 / 95	0 / 105
Vulnerable Situations	3	0	0

this by having three types of parameters and trying to see which ones can be added to automate the process in advance. Similarly, we cannot delete individuals or assertions. Again - this could be potentially done by code (e.g., if such an assertion comes from the rule, the existing one should be deleted). It would be exciting to explore the possibility of SWRL, which could update the assertions. However, this could introduce many inconsistencies and complexities for the reasoner. We are still debating the best way to present the policies, which sends notifications about the changes (e.g., letter about changed address).

## 6 Conclusions and Future Work

In this paper, we presented the analysis on the third level of our framework. The main result of this work is the dynamic approach, where we incorporated a reasoner in a dynamic matter where output becomes an input, and at the same time, this loop is finite since the model is finite. Our approach can model the user-based application’s policies and find logical vulnerabilities in the identification-authentication- authorization policies.

It would be interesting to explore the possibility of retrieving the triggered rules by the reasoner. However, at the moment, it seems hardly possible even though all action names are unique. We have one more concept, which could be

introduced in future work. It is a class Event concept, representing the system's occurrences, such as broken connections, expired sessions etc.

**Acknowledgments.** This research has been partially funded by Lloyds Banking Group in London, UK. However, the results and the opinions formulated in the paper are the author's only. No actual data from the bank has been used in the paper.

**Disclosure of Interests.** The authors have no competing interests to declare that are relevant to the content of this article.

## References

1. Agrawal, V.: Towards the ontology of iso/iec 27005: 2011 risk management standard. In: International Symposium on Human Aspects of Information Security and Assurance (2016)
2. Bataityte, K., Vassilev, V., Gill, O.J.: Ontological foundations of modelling security policies for logical analytics. In: Artificial Intelligence Applications and Innovations (06 2020)
3. Fenz, S., Ekelhart, A.: Formalizing information security knowledge. In: Proceedings of the 4th International Symposium on Information, Computer, and Communications Security. p. 183–194. ASIACCS '09, Association for Computing Machinery, New York, NY, USA (2009). <https://doi.org/10.1145/1533057.1533084>, <https://doi.org/10.1145/1533057.1533084>
4. Herzog, A., Shahmehri, N., Duma, C.: An ontology of information security. IJISP **1**, 1–23 (10 2007). <https://doi.org/10.4018/jisp.2007100101>
5. Horrocks, I., F. Patel-Schneider, P., Boley, Haroldand Tabet, S., Grosf, B., Dean, M.: Swrl: A semantic web rule language combining owl and ruleml. <https://www.w3.org/Submission/SWRL/> (May 2004)
6. Moreira, E., Martimiano, L., Brandão, A., Bernardes, M.: Ontologies for information security management and governance. Information Management and Computer Security **16**, 150–165 (06 2008). <https://doi.org/10.1108/09685220810879627>
7. Parkin, S., van Moorsel, A., Coles, R.: An information security ontology incorporating human-behavioral implications. pp. 46–55 (01 2009). <https://doi.org/10.1145/1626195.1626209>
8. Penelova, M.: Access control models. Cybernetics and Information Technologies **21**, 77–104 (12 2021). <https://doi.org/10.2478/cait-2021-0044>
9. Ramanauskaitė, S., Olifer, D., Goranin, N., Cenys, A.: Security ontology for adaptive mapping of security standards. International Journal of Computers, Communications and Control (IJCCC) **8**, 813–825 (11 2013). <https://doi.org/10.15837/ijccc.2013.6.764>
10. Souag, A., Salinesi, C., Mazo, R., Wattiau, I.: A security ontology for security requirements elicitation. In: Engineering Secure Software and Systems (03 2015). [https://doi.org/10.1007/978-3-319-15618-7\\_13](https://doi.org/10.1007/978-3-319-15618-7_13)
11. Szeredi, P., Lukácsy, G., Benkő, T.: The Semantic Web Explained: The Technology and Mathematics Behind Web 3.0. Cambridge University Press, New York, NY, USA (2014)