Towards Automated Learning of Access Control Policies Enforced by Web Applications

Padmavathi Iyer
University at Albany – SUNY
Albany, New York, USA
riyer2@albany.edu

Amir Masoumzadeh
University at Albany – SUNY
Albany, New York, USA
amasoumzadeh@albany.edu

ABSTRACT

Obtaining an accurate specification of the access control policy enforced by an application is essential in ensuring that it meets our security/privacy expectations. This is especially important as many of real-world applications handle a large amount and variety of data objects that may have different applicable policies. We investigate the problem of automated learning of access control policies from web applications. The existing research on mining access control policies has mainly focused on developing algorithms for inferring correct and concise policies from low-level authorization information. However, little has been done in terms of systematically gathering the low-level authorization data and applications' data models that are prerequisite to such a mining process. In this paper, we propose a novel black-box approach to inferring those prerequisites and discuss our initial observations on employing such a framework in learning policies from real-world web applications.

CCS CONCEPTS

• **Security and privacy** → **Access control**; **Authorization**.

KEYWORDS

policy mining, web application, relationship-based access control, automated, concrete systems

ACM Reference Format:
Padmavathi Iyer and Amir Masoumzadeh. 2023. Towards Automated Learning of Access Control Policies Enforced by Web Applications. In *Proceedings of the 28th ACM Symposium on Access Control Models and Technologies (SACMAT '23), June 7–9, 2023, Trento, Italy.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3589608.3594743

1 INTRODUCTION

As applications adopt more complex access control policies, it becomes harder to provide accurate specifications of the enforced policies. Such an accurate specification is vital both for verifying the security/privacy provisions in an application and for offering a clear expectation of those provisions to end users. In this paper, we explore whether we can learn access control policies from an existing application. As human users, it is intuitive that when we work with an application we can make inferences about its enforced access control policies. However, we note that such inferences can take time to form and may not be necessarily accurate. Is it possible to develop a systematic methodology that automates such a learning process in order to infer a comprehensive access control policy enforced by an application? This is the question that we explore more specifically in this paper in the context of web applications which constitute a majority of everyday applications nowadays. In this context, we consider learning relationship-based access control policies (ReBAC) [8, 11, 19] that support expressing authorization in terms of relationship patterns between application entities.

While approaches to mining access control policies [5, 7, 16, 21] are useful in addressing the above problem, we note that they can only provide a part of the solution. Policy mining focuses on inferring high-level policies in a target domain from existing lower-level authorization information (e.g., access control lists). This process also often requires access to a model of application data. For example, a ReBAC policy miner typically requires two inputs: a system graph that contains information about entities and their relationships and a set of lower-level authorizations implemented in the application. In the literature, ReBAC miners have assumed the provision of both inputs, and have primarily focused on developing efficient approaches for inferring correct and concise policies. However, to realize our goal of learning policies from real-world applications, we need to systematically gather both of those inputs, preferably in an automated fashion. Furthermore, we propose to approach these problems in a black-box manner, i.e., making inferences through observing user interactions with the application. Such an approach assumes a minimal dependency on web protocols, and is in contrast to developing white-box approaches that have to depend heavily on the availability and specific languages/models of application internals (e.g., server-side source code, database, etc.).

In this paper, we present our preliminary results in developing a black-box framework for learning web application access control policies by automated inference of the data model of applications and observing applicable authorizations to that data. We develop algorithms to infer fine-grained attributes of application data objects as they are communicated between a web application's client and server code (i.e., web requests/responses). We also develop techniques to re-identify previously inferred data objects and their attributes in a web application's pages to make an accurate observation of applicable authorizations. Our experimental results on two real-world applications show the feasibility of our framework.

2 BACKGROUND

An application comprises users, resources, and logical components that may be related to one another based on contextual information.

SACMAT '23, June 7–9, 2023, Trento, Italy

ACM ISBN 979-8-4007-0173-3/23/06…$15.00
https://doi.org/10.1145/3589608.3594743

We employ the *relationship-based access control* (ReBAC) [8, 11, 19] model to specify authorizations in such an application with a rich data model. To capture the authorization state, we employ an *attributed system graph* that represents the relationship data among application entities as well as the entity attributes as a directed attributed graph. Suppose L is the set of relationship types. Then, we denote a system graph as $G = \langle V, E, \eta \rangle$. Here, V is the set of nodes in the system graph, which we refer to as *data objects*. Data objects correspond to application entities such as users, resources, and logical components. $E \subseteq V \times V \times L$ represent various labeled (from L) relationships that exist between the data objects. The attribute assignment η is a function that takes a data object $v \in V$ as input and produces the set of attributes associated with v, which we denote as $\eta\ (v)$. We represent $\eta\ (v)$ as a set of key-value pairs $\{\alpha_1 : d_1, \alpha_2 : d_2, \ldots, \alpha_n : d_n\}$, where α_i is an attribute name and d_i is the corresponding data value. We use dot notation to indicate an element within a concept (e.g., $G.V$ refers to system graph nodes).

To form meaningful system graphs, we employ a *graph schema*, $\mathcal{G} = \langle \mathcal{V}, \mathcal{E} \rangle$, to constrain the types of nodes and edges. Here, $\mathcal{V} = \{\tau_1, \tau_2, \ldots, \tau_n\}$ indicates a set of *data types*. Each data type τ defines a group of data objects $V_\tau = \{v_1, v_2, \ldots, v_n\}_\tau$ that have some shared attributes. So, users, resources, and each logical entity are assigned different data types. $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V} \times L$ constraints the relationships that can exist between two data objects. In the above notations, we use an abstract character $n \geq 1$ to indicate the upper bounds of the corresponding elements. Let function $\tau(v)$ return the type of a data object. Then, for every $v \in V$, it holds that $\tau(v) \in \mathcal{V}$. Also, for every relationship $\langle v_i, v_j, l \rangle \in E$, it holds that $\langle \tau(v_i), \tau(v_j), l \rangle \in \mathcal{E}$. We say such a system graph G is *well-formed* with respect to schema $\mathcal{G}$. The well-formedness is implied when we discuss system graphs.

The ReBAC model controls access based on a set of rules. The ReBAC rules employ *relationship patterns* that specify different arrangements of labeled edges between the entities in a system graph. We denote a relationship pattern ϕ as a sequence of relationship labels $[l_1, l_2, \ldots, l_n]$, where $l_i \in L$ and $n \geq 1$. We use $-l$ to represent an edge with label $l \in L$ traversed in the inverse direction. A *ReBAC policy* $\rho = \{\phi\ [\wedge\ \phi]\}$ consists of a set of ReBAC authorization rules, where each rule is a conjunction of a set of relationship patterns. E.g., the rule "$[\mathit{friend}, \mathit{friend}, \mathit{owns}] \wedge [\mathit{city}, -\mathit{city}, \mathit{owns}]$" allows access by only friends of friends living in the same city as the owner. An *access request*, $\langle u, r \rangle \in V \times V$, will be *permitted* if it matches a rule in ρ, otherwise denied. A request $\langle u, r \rangle$ matches a rule if it matches every pattern ϕ in the rule, i.e., there is a path from u to r in G such that the sequence of edge labels in the path matches the sequence of the labels in ϕ. Alternatively, we say ϕ *applies* to $\langle u, r \rangle$ in such a case. We can characterize a ReBAC policy that is enforced in an application by calculating the access decision δ corresponding to every combination of $\langle u, r \rangle$. We refer to such a collection of $\langle u, r, \delta \rangle$ as *lower-level authorizations*, denoted as $\mathcal{Z}$.

3 OVERVIEW OF PROBLEM & SOLUTION

We learn the ReBAC policy from a web application in a black-box manner. To this end, we observe the user interactions with a web application and analyze the application's responses to various user operations. A user can navigate through an application to view different data objects or can create a data object by inputting its details. The client machine and the application server need to communicate to render appropriate content on the user interface. We collect such client-server communication (i.e., web requests/responses) whenever a user interacts with an application, which includes user-generated content, the data that the client provides to the server, and the data generated by the server in response to client data. We refer to such a collection of web-based interactions as *traces*.

We generate the authorizations corresponding to what users can view on an application page both at coarse and fine-grained levels. Specifically, we consider two types of policies that control access to application data, namely *object-level policy* and *attribute-level policy*. The former specifies what data objects can a user view on a webpage (e.g., Alice can view Bobby's post but not Carol's), while the latter specifies what attributes of a visible data object are accessible to a user (e.g., Alice can view only the name of Bobby's post but Bobby can view the location of the post as well). We produce the set of relationship patterns for both these authorization levels.

Our proposed framework undertakes a two-fold approach. First, it observes the various user operations that result in accumulating data in a web application (by inspecting the web requests/responses) to infer a model of the application data encompassing the entities and their relationships. Second, using the inferred data model, our approach determines the lower-level authorizations that are applicable to application data by observing the resources that users can access on application pages. Those include authorization information on fine-grained data object attributes for application resources (e.g., title, author, and time for a post in an online social network).

We remark on some of the assumptions that we make in our solutions. First, we can observe the complete set of lower-level authorizations enforced in a web application. If a user can view a data object on any page, that user is permitted to access the object. If a user is not able to view a data object on any page of the application, then we consider the user as being denied access to that object. Secondly, for the convenience of navigating and exploring various data objects, we assume that we have listings pages in a web application. We refer to a listings page as a web page that displays a list of all existing resources of one type. Such listings pages existed in the web applications that we experimented on and an expert who has knowledge of the application provided us with such listings pages for every type of resource. Finally, the lower-level authorizations enforced over existing resources do not invalidate during a complete run of our framework. The entities can be added as well as relationships can be formed during the creation of entities. But, the authorizations over existing resources cannot change when the application has new additions.

4 BLACK-BOX DATA MODEL INFERENCE

In this section, we describe our process for inferring a model of the data implemented in a web application, which includes: 1) data types $\mathcal{V}$, 2) data objects V_τ for every $\tau \in \mathcal{V}$, 3) data object attributes $\eta\ (v)$ associated with every $v \in V$, 4) relationship constraints $\mathcal{E} = \{\langle \tau(v_i), \tau(v_j), l \rangle\}$ where $\tau(v_i)$ and $\tau(v_j)$ are members of $\mathcal{V}$, and 5) relationships between data-object pairs $E = \{\langle v_i, v_j, l \rangle\}$ where $v_i \in V_{\tau(v_i)}$ and $v_j \in V_{\tau(v_j)}$. Our methodology functions in a black-box manner by collecting web traces. Although we need only the system graph to carry out the policy mining task, we are also inferring its schema because it plays an important role in determining the relationships that can exist in the corresponding system graph.

4.1 Obtaining Information about Data Objects

Initially, we parse the given web interaction traces and obtain the potential data object attributes associated with each *request*. Then, we merge the obtained attributes to determine different data types.

Parsing Request/Response Contents in Traces. During the data-model inference, we refer to each request γ in the given traces as an instance of a data-object creation. In other words, each request corresponds to a data object, and so the properties associated with the request can be used to characterize the corresponding data object. We divide the set of object attributes that we need to infer from the given traces into client-generated, including user-generated, and application-generated properties. To capture both kinds of these properties, we utilize especially two kinds of information in our traces for every γ, namely, request content Q_γ that comprises information provided by an end-user when creating a certain type of data object and response content $\mathcal{S}_\gamma$ that comprises information that is generated by the application when a data object is created in order to identify the object during future retrievals. For each instance of data-object creation, we parse Q_γ to obtain the values provided by a user for different fields on the creation page and append them to $\mathcal{A}_\gamma$ as a set of "field-value" pairs. Parsing Q_γ also reveals client-provided values that complement user-provided values, such as the ID of the post associated with a comment. Furthermore, we parse $\mathcal{S}_\gamma$ to obtain application-generated values. For instance, for a parametric response such as JSON, we simply append the key-value pairs in the content to $\mathcal{A}_\gamma$. On the other hand, for a listing-based response such as HTML, we calculate the difference in the page content before and after the addition of the data object to the page. In this way, we can differentiate the values that are *generated* by the application from static, styling-oriented elements on a web page.

Pruning & Merging Data Object Attributes. We want to prune all spurious data object attributes that are not relevant to real data such as styling elements that are introduced when a data object is added to a page. Additionally, we want to observe commonalities between the attributes of various requests γ and group them into different data types based on the similarity of their attributes. To this end, we perform a bottom-up hierarchical clustering, where each request γ is initially placed in a separate cluster. In each iteration, we merge the clusters whose object attributes have the highest similarity, until the number of clusters equals an expert-provided value. For every pair of requests γ_i and γ_j, we measure the similarity based on the number of common keys $\{\alpha\}$ and their corresponding values $\{d\}$ in $\mathcal{A}_{\gamma_i}$ and $\mathcal{A}_{\gamma_j}$. Thus, our approach produces a set of clusters that each represent a different data type τ in our data model and each request in a cluster τ corresponds to a data object v of that type, i.e., $v \in V_\tau$. Moreover, the attributes that are repeating across all the objects within a cluster τ can be safely ignored from consideration since they do not add any specific knowledge about the respective data objects, and we obtain $\eta\ (v)$ for every $v \in V_\tau$. For example, we remove the "class" property if it is used for styling all objects of a particular type, but retain the "id" property.

4.2 Inferring Data-Object Relationships

The pruned and merged graph that was produced still constitutes an unpolished data model because of the absence of relationships. To determine the relationships that can exist between a pair of data types, we observe the commonalities between their respective data objects, based on their attributes that we inferred previously. There are two ways in which we determine the connections between a pair of data objects – 1) when both objects have the same values for some attribute, i.e., $\exists \alpha_i, \alpha_j$, it holds that $\eta(v_i).\alpha_i = \eta(v_j).\alpha_j$, and 2) when one object contains the value or a part of the value associated with another object. If we visualize the above process in terms of a graph, then, in the former case, we obtain the relationship pattern $[\alpha_i, -\alpha_j]$ from v_i to v_j for each data value that is shared between both objects. (Recall that we use the notation $-l$ to traverse an edge $\langle v_1, v_2, l\rangle$ in the reverse direction.) For instance, if comments are present on the same page as their post, then we obtain a pattern $[URL, -URL]$ that connects individual posts with their comments.

In the latter case, we again have two cases. Suppose the value of an attribute, say "URL", for a comment object v_i is a query string, e.g., "*https://example.com/comments?postid=1&postname=sample-post*". Then, we parse the query string to obtain values "1" and "sample-post", which are the values of attributes, say "id" and "name", associated with a post object v_j. So, we obtain relationship patterns $[URL, contains, -id]$ and $[URL, contains, -name]$ from a comment to its post. Otherwise, for data values other than query strings, we segregate the common and distinct components within the values of some attribute associated with all data objects of a particular type and determine if the distinct components are equal to or are contained within some other objects' values. For instance, an attribute "container-id" associated with comments has values "object-1", "object-2", "object-3". Observe that "object" is common between all values. If the posts in the application have an attribute called "id" with values "1", "2", "3", then we obtain the pattern $[container\text{-}id, contains, -id]$ from a comment to its respective post.

Thus, our approach produces a mapping $\{\phi : [\langle v_i, v_j\rangle]\}$ between relationship patterns and the pairs of data objects that are connected by the corresponding pattern. Relationships exist between two different data objects (because we do not consider loops in our system graphs) of either the same type (e.g. friend relationship between two users) or two different types. Subsequently, we consider that the relationship patterns that we calculated by linking the attributes between two data objects correspond to *potential* relationships types between those objects. Specifically, for each relationship pattern ϕ in $\{\phi : [\langle v_i, v_j\rangle]\}$, we add a relationship $\langle v_i, v_j, l\rangle$ into E, where l is an abstract label that is unique for each ϕ and $(v_i \in V_{\tau_i}, v_j \in V_{\tau_j})$. Additionally, for every $\langle v_i, v_j\rangle$, we add the edge $\langle \tau(v_i), \tau(v_j), l\rangle$ into $\mathcal{E}$. Finally, we prune redundant relationships by employing two kinds of heuristics: 1) removing relationships that exist between all pairs of data objects (irrespective of the data type) and 2) removing relationships that can be derived from other relationships. We want to ensure that, similar to data object attributes, we find meaningful object relationships and remove any "noise" that can be caused when there are different semantics associated with the same artifact. Consequently, to ensure that E is well-formed with respect to $\mathcal{E}$, we remove the edges $\langle v_i, v_j, l\rangle$ corresponding to each pruned relationship constraint $\langle \tau(v_i), \tau(v_j), l\rangle$, where $(v_i \in V_{\tau_i}, v_j \in V_{\tau_j})$.

5 INFERRING LOW-LEVEL AUTHORIZATIONS USING INFERRED DATA MODEL

Previously, we presented our approach for mining an application's data model. Our goal is to mine the ReBAC policy enforced in the application. Mining ReBAC policies requires two inputs, namely the data model and the complete set of lower-level authorizations (§ 3). So, in this section, we tackle the problem of mining lower-level authorizations that, in our context, specify if a user can access a data object and its associated attributes. We use inferred data model along with a set of observations in the form of web traces.

5.1 Generating Traces for Authorization Mining

To mine the lower-level authorizations, we need traces that can describe the contents of different pages relative to different users when they are logged into a web application. Suppose $\mathcal{U}$ be the set of users in an application and $\mathcal{P}$ be all the pages that users can navigate to. Then, in the context of authorization mining, we denote the traces $\mathcal{T}_\gamma$ corresponding to a request γ as $\langle u, p, \mathcal{S}_\gamma \rangle$ where the target application produces content $\mathcal{S}_\gamma$ when $u \in \mathcal{U}$ navigates to $p \in \mathcal{P}$. The content $\mathcal{S}_\gamma$ associated with a request will then be used for re-identifying data objects and their attributes present in $\mathcal{S}_\gamma$.

To precisely mine an application's policy, a miner requires complete knowledge of the lower-level authorizations. So, we need to have a request for every user navigating to every accessible page. If a user can view a data object and its attributes on any of those pages, then we regard the user as being *permitted* to access the object. Otherwise, the user is *denied* access to the object. We exploit the notation $\mathcal{Z}$ from § 2 to represent the authorizations for a given user $u \in \mathcal{U}$ as $\mathcal{Z}(u)$ and to represent the authorizations for a given user u and the content $\mathcal{S}_\gamma$ in response to a request submitted by the user as $\mathcal{Z}(u, \mathcal{S}_\gamma)$. Thus, we calculate the complete set of authorizations for each user as $\mathcal{Z}(u) = \bigcup_{p \in \mathcal{P}} \mathcal{Z}(u, \mathcal{S}_\gamma)$, where $\mathcal{T}_\gamma = \langle u, p, \mathcal{S}_\gamma \rangle$. Finally, we repeat the above process for every user u in $\mathcal{U}$.

5.2 Re-identifying Data Objects from Traces

We determine if an object in our data model $v \in V_\tau$, for some τ, is present in content $\mathcal{S}_\gamma$. But, to decide if some data object exists in the content of some traces, we must have some means to *re-identify* that particular object out of all the information present in the content. Our approach comprises two steps, namely calculating the identifying information about each data object, which we also refer to as *anchors*, and then re-identifying the data objects and their attributes from the traces by using the obtained anchors.

Mining Anchors from Data Model. There is no abstract notion of data objects in the real data as found in traces. A data object is usually characterized by a set of attributes. Therefore, in order to re-identify a data object $v_i \in V_\tau$, $\exists \tau \in \mathcal{V}$, from some traces, we must basically check for the existence of its associated attributes $\eta(v_i)$ in the traces' content $\mathcal{S}_\gamma$. More specifically, we need to determine the unique values $[d_i] \subseteq \eta(v_i).\alpha_i$, $\forall \alpha_i$, that can distinguish v_i from all other $v \in V$. (The comparison between data values for determining uniqueness is based on matching the attribute name as well as its value.) We refer to such a set of data values that can be used to uniquely identify a data object from the contents of traces as *anchors*. The anchors corresponding to a data object provide a means to identify if the object is present in some content $\mathcal{S}_\gamma$. We have no prior knowledge about the contents of a page in the application that a user can navigate to. We can have any type of as well as any number of data object(s) on a certain page when visited by a certain user. Moreover, the set of traces given to us $\mathcal{T}_\gamma$, for every γ, is simply a collection of observations about what different users are authorized to see in the application. As a result, there is no ordering between data objects or provision of any other metadata as such that can aid in pointing out an object from a traces' content.

Observing Lower-Level Authorizations using Anchors. We utilize the anchors in our data model to, initially, determine what objects and their attributes a user is authorized to view on a given page of the application, i.e, $\mathcal{Z}(u, \mathcal{S}_\gamma)$, when the user is logged in. For each data object that is present in page content $\mathcal{S}_\gamma$, we now need to determine whether the data values associated with the object, other than those included in the anchors, are present in $\mathcal{S}_\gamma$. To this end, we first figure out the order in which data objects appear in $\mathcal{S}_\gamma$, by retrieving the positions of the first and last occurrences of the objects' anchors. Assuming that the values of different data objects do not completely overlap in $\mathcal{S}_\gamma$, we employ a "rolling window" to obtain a portion of $\mathcal{S}_\gamma$ that potentially encompasses all the attribute values corresponding to a visible object. We consider the boundary of the window corresponding to a data object to be between the last occurrence of the previously placed object's anchor and the first occurrence of the immediately next object's anchor. Finally, to determine the set of lower-level authorizations for every user, i.e., $\mathcal{Z}(u)$, we aggregate all the data objects and their attributes that are visible to the currently logged-in user on any page of the target application. That is, by repeating for each $p \in \mathcal{P}$ and $u \in \mathcal{U}$, and aggregating the visible data elements, we obtain our desired result.

6 IMPLEMENTATION AND EVALUATION

We performed experiments on two social network web applications, namely Elgg [1] and Funkwhale [3], to evaluate our approach. Elgg allows users to add friends, create photo albums as well as view their friends' albums and comment on them. Funkwhale is a music-sharing application that allows users to create libraries and upload audio files to them as well as access remote libraries. We employed a state-of-the-art ReBAC mining algorithm from the literature [13] and mine a ReBAC policy for each system based on the system graph and lower-level authorization inferred by our prototype. We first describe our experimental setup followed by observations.

6.1 Setup and Configurations

We employed *mitmproxy* [4] in our prototype implementation for capturing the HTTP conversations between client and server. We executed every experiment 10 times and reported the average result.

6.1.1 Experiment Procedure & User Simulation. Our implementation first crawls the web application simulating operations like a normal user and collects HTTP traces, which are inputted into our algorithm for mining the data model. Then, our implementation crawls through every page, while logging in as different users, and the collected HTTP traces, along with the data model, are used for mining the access log. Finally, our implementation executes the

Table 1: Evaluation of Mined Attribute-Level Policy Based on Different Rules and Data Types. (Meaning of Relationships in Rules: b=belongs-to-post, c=commented, f=friends-with, i=in-library, o=owns-library, p=posted, w=follows-library.)

Application	Rule	Data Type	Mined	Ground-Truth (visible atts.)	$\lvert\Updownarrow\rvert$	$\lvert\Uparrow\rvert$	$\lvert\not\Uparrow\rvert$	$\lvert\Downarrow\rvert$
Elgg	$[p]$	Post	27	19	16	6	5	3
	$[f, p]$	Post	23	15	12	6	5	3
	$[p, -b]$	Comment	20	12	11	3	6	1
	$[c]$	Comment	24	16	15	3	6	1
	$[f, p, -b] \wedge [f, c]$	Comment	20	12	11	3	6	1
Funkwhale	$[o, -i]$	Audio file	35	28	24	6	5	4
	$[w, -i]$	Audio file	29	23	20	6	3	3

ReBAC miner while inputting the inferred data model and authorizations, and the mined policy is presented for expert evaluation.

To generate request-response traces, we use *UI.Vision RPA* Web Driver [2] for user simulation which facilitates submitting music and image files during the crawling process. We provide the crawler with the set of user credentials and possible atomic actions (e.g., posting a photo). The crawler then logs into an application and randomly selects an operation among the set of available atomic operations for the currently chosen user. Our crawler initially logs into the admin account and creates a set of users for each application – 10 users for Elgg and 15 users for Funkwhale. Throughout the crawling phase, mitmproxy captures HTTP dumps of these simulated interactions. Our crawling process includes 3 data types, about 200 data objects, and about 2000 relationships for each application.

6.1.2 Evaluation Methodology. Our evaluation is in terms of the correctness of mined policy at both the object level and attribute level. That is, in each application and for each data object type, we compare the object attributes that are visible in an authorized user's interface with those specified in the mined policy. Also, based on the mined attributes, we make observations about the authorizations of data objects associated with those attributes. For ground truth, we rely on a manual inspection of the application. For assessing object-level policies, we employ an application's documentation as the ground truth. For assessing attribute-level policies, a human expert determines fine-grained visibility of object attributes in the application interface by logging in as users with different relationship patterns (e.g., logging in as owner). The expert may also inspect an application's database for verifying the data objects and their attributes. We emphasize that the ground-truth policy or access to the database is used only for evaluation purposes.

6.2 Observations

The second column of Table 1 shows the rules in our mined object-level policy for both Elgg and Funkwhale applications. By checking against the ground truth, we observed that our approach was able to capture all the authorizations enforced on data objects, i.e., our object-level policy is precise. To gain better insight into the performance of mining attribute-level policies, we categorize the mined rules into: 1) those that are correct with respect to the ground truth, 2) those that are not present in the ground truth but not incorrect, 3) those not present in the ground truth and incorrect, and 4) those that are present in the ground truth but are not in the mined policy. These four categories are shown in the last four columns of the

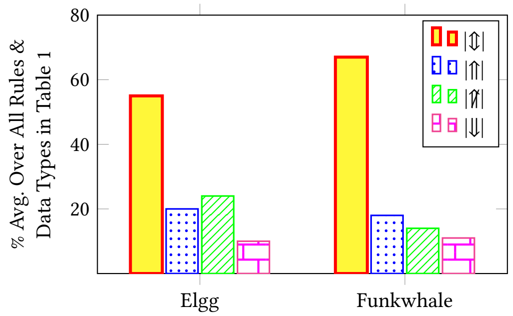

Figure 1: Visual Comparison of the Performance of Mining Attribute-Level Policies for Elgg and Funkwhale Applications

table using the notations $|\Updownarrow|$, $|\Uparrow|$, $|\not\Uparrow|$, and $|\Downarrow|$, respectively, along with the mined attribute-level rules and the ground-truth attribute-level policy. Each row in the table represents, for a given object-level rule, what attributes of a certain type of data object are accessible (i.e., visible) to the users who satisfy the pattern in the rule. Below we summarize our results based on the four performance categories:

$|\Updownarrow|$ For both applications, this is the dominating category. We are able to mine most of the authorizations enforced on data object attributes that are visible to users satisfying the corresponding relationship pattern with a data object.

$|\Uparrow|$ This category is often comprised of fewer cases that seem like extra inferences. But a closer comparison with the application database reveals that such attribute values describe identifying characteristics of an object such as its id, date-time, and parent-id, and are usually server-generated.

$|\not\Uparrow|$ This category comprises extra inferences that do not add any real meaning to data objects. Such attributes are neither client- nor server-generated; they mostly cover style components and other metadata that serve the purpose of giving a general description of the application and/or its data objects.

$|\Downarrow|$ This category happens due to the constant update of content containing certain attributes, usually to enhance the user experience. For instance, the date-time in Elgg is displayed as "just now" when creating an object but gets updated to, say, "2 minutes ago" when generating authorization traces.

Although not applied in the above results, it is possible to work around the classes of $|\Uparrow|$ and $|\not\Uparrow|$. The contents of HTML tags usually result in displaying data on the user interface. So, if we annotate those object attributes that are tag contents as visible, then during authorization mining we can simply focus on the visible attributes and ignore the others. Similarly, it is possible to develop heuristics to deal with the class of $|\Downarrow|$, but we chose not to for maintaining the generality of our approach and to manifest the existing challenges.

Figure 1 presents a visual demonstration of Table 1. The results for both Elgg and Funkwhale are shown side-by-side on the x-axis. The y-axis measures, for each application, the number of attribute-level rules that fall into each of the four performance categories averaged over all the object-level rules and data types, which are individually described in the table. We can observe that, for both applications, we were able to mine most of the attributes associated with all data objects. Interestingly, the number of correct attributes are more for Funkwhale than Elgg. From a technology standpoint, Funkwhale uses extensive AJAX communications with its web server to handle insertion requests/responses and render the user interface. The AJAX response contains JSON data, i.e., a list of key-value pairs that we consider as application-generated content. So, we could avoid retrieving extra components such as style elements when inferring attributes directly from an application's content.

7 RELATED WORK

Mining Access Control Policies. This problem has been widely studied for role-based access control (RBAC) [16, 17], attribute-based access control (ABAC) [7, 12, 21], and more recently for relationship-based access control (ReBAC) [5, 6, 13]. These works assume that lower-level authorizations and information about users and resources are already provided for the mining task. In contrast, we treat a web application as a black box and mine its enforced ReBAC policy by simply observing human interactions with the application. By collecting web traces, we infer the data model comprising relationships among users and resources as well as infer the lower-level authorizations comprising what users can see on an application page. Also, using the mined data model and authorizations, we produce two levels of policies – a course-grained object-level policy that controls access to data objects and a fine-grained attribute-level policy that controls access to object attributes.

Black-Box Vulnerability Detection. The problem of detecting vulnerabilities in web applications has been widely studied. This includes alleviating data disclosure vulnerabilities by monitoring HTTP traffic for users' data items [18], validating the enforcement of access control policies that are not clearly documented [14], and effectively crawling a web application by considering its internal state to discover more vulnerabilities [9]. Additionally, researchers have focused on identifying logic vulnerabilities based on different user interaction patterns [20] as well as on detecting state violation attacks by extracting invariants and session values from interactions between clients and stateless web application [15]. More recently, the problem of employing differential traffic analysis for vulnerability detection in mobile apps has been considered [22], along with enhancing black-box crawling and scanning of web applications to detect cross-site scripting vulnerabilities [10]. From a black-box analysis perspective, while these prior works focused on behavioral patterns of user interaction and navigation structure of web application pages, our focus is on the users' data objects themselves including recognizing uniquely identifying resource parameters and different kinds of associations that can be inferred from them in the context of multi-user applications.

8 CONCLUSION

We proposed a methodology to infer the enforced ReBAC policy rules in a web application by observing client-server interactions during the creation of data objects and their visibility in the application as viewed by different users. Our experiments on two social networking applications demonstrated precise mining of object-level policies in both cases. We were also able to mine the attribute-level policies fairly accurately considering their visibility in each application's interface. As future work, we plan to investigate the efficacy and further automation of our trace generation, and expand our experiments to a larger set of real-world applications.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No. 2047623.

REFERENCES

[1] 2004. elgg. https://elgg.org/.

[2] 2016. UI.Vision RPA. https://ui.vision/kantu.

[3] 2017. Funkwhale. https://funkwhale.audio/en_US/.

[4] 2019. mitmproxy. https://mitmproxy.org/.

[5] Thang Bui and Scott D Stoller. 2020. A Decision Tree Learning Approach for Mining Relationship-Based Access Control Policies. In *Proceedings of the 25th ACM Symposium on Access Control Models and Technologies*. 167–178.

[6] Thang Bui, Scott D Stoller, and Jiajie Li. 2019. Greedy and evolutionary algorithms for mining relationship-based access control policies. *Computers & Security* 80 (2019), 317–333.

[7] Carlos Cotrini, Thilo Weghorn, and David Basin. 2018. Mining ABAC rules from sparse logs. In *IEEE European Symposium on Security and Privacy*. IEEE, 31–46.

[8] Jason Crampton and James Sellwood. 2014. Path conditions and principal matching: a new approach to access control. In *Proc. ACM SACMAT*. 187–198.

[9] Adam Doupé, Ludovico Cavedon, Christopher Kruegel, and Giovanni Vigna. 2012. Enemy of the state: A state-aware black-box web vulnerability scanner. In *21st USENIX Security Symposium (USENIX Security 12)*. 523–538.

[10] Benjamin Eriksson, Giancarlo Pellegrino, and Andrei Sabelfeld. 2021. Black widow: Blackbox data-driven web scanning. In *IEEE S&P*. IEEE, 1125–1142.

[11] Philip WL Fong. 2011. Relationship-based access control: protection model and policy language. In *Proc. ACM CODASPY*. 191–202.

[12] Padmavathi Iyer and Amirreza Masoumzadeh. 2018. Mining positive and negative attribute-based access control policy rules. In *Proceedings of the 23nd ACM on Symposium on Access Control Models and Technologies*. 161–172.

[13] Padmavathi Iyer and Amirreza Masoumzadeh. 2019. Generalized Mining of Relationship-Based Access Control Policies in Evolving Systems. In *Proc. 24th ACM SACMAT*. ACM, 135–140.

[14] Ha Thanh Le, Cu Duy Nguyen, Lionel Briand, and Benjamin Hourte. 2015. Automated inference of access control policies for web applications. In *Proc. of the 20th ACM Symposium on Access Control Models and Technologies*. ACM, 27–37.

[15] Xiaowei Li and Yuan Xue. 2011. BLOCK: a black-box approach for detection of state violation attacks towards web applications. In *Proceedings of the 27th Annual Computer Security Applications Conference*. ACM, 247–256.

[16] Barsha Mitra, Shamik Sural, Jaideep Vaidya, and Vijayalakshmi Atluri. 2016. A survey of role mining. *ACM Computing Surveys (CSUR)* 48, 4 (2016), 1–37.

[17] Ian Molloy, Ninghui Li, Yuan Alan Qi, Jorge Lobo, and Luke Dickens. 2010. Mining roles with noisy data. In *Proc. 15th ACM SACMAT*. ACM, 45–54.

[18] Divya Muthukumaran, Dan O'Keeffe, Christian Priebe, David Eyers, Brian Shand, and Peter Pietzuch. 2015. FlowWatcher: Defending against data disclosure vulnerabilities in web applications. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 603–615.

[19] Edelmira Pasarella and Jorge Lobo. 2017. A datalog framework for modeling relationship-based access control policies. In *Proc. ACM SACMAT*. 91–102.

[20] Giancarlo Pellegrino and Davide Balzarotti. 2014. Toward Black-Box Detection of Logic Flaws in Web Applications.. In *NDSS*.

[21] Zhongyuan Xu and Scott D Stoller. 2014. Mining attribute-based access control policies. *IEEE Trans. on Dependable and Secure Computing* 12, 5 (2014), 533–545.

[22] Chaoshun Zuo, Qingchuan Zhao, and Zhiqiang Lin. 2017. Authscope: Towards automatic discovery of vulnerable authorizations in online services. In *Proc. 2017 ACM Conference on Computer and Communications Security*. ACM, 799–813.