Automated Inference of Access Control Policies for Web Applications

Conference Paper · June 2015

DOI: 10.1145/2752952.2752969

4 authors, including:

Ha Thanh Le
University of Luxembourg
11 PUBLICATIONS **52** CITATIONS
SEE PROFILE

Cu Duy Nguyen
Post Luxembourg
47 PUBLICATIONS **913** CITATIONS
SEE PROFILE

Benjamin Hourte
EarthLab Luxembourg S.A.
4 PUBLICATIONS **14** CITATIONS
SEE PROFILE

Automated Inference of Access Control Policies for Web Applications

Ha Thanh Le, Cu D. Nguyen, and Lionel Briand
Interdisciplinary Centre for Security, Reliability and Trust (SnT Centre)
University of Luxembourg, Luxembourg
{hathanh.le,duy.nguyen,lionel.briand}@uni.lu

Benjamin Hourte
HITEC Luxembourg S.A.
L-1458 Luxembourg
benjamin.hourte@hitec.lu

ABSTRACT

In this paper, we present a novel, semi-automated approach to infer access control policies automatically for web-based applications. Our goal is to support the validation of implemented access control policies, even when they have not been clearly specified or documented. We use role-based access control as a reference model. Built on top of a suite of security tools, our approach automatically exercises a system under test and builds access spaces for a set of known users and roles. Then, we apply a machine learning technique to infer access rules. Inconsistent rules are then analysed and fed back to the process for further testing and improvement. Finally, the inferred rules can be validated based on pre-specified rules if they exist. Otherwise, the inferred rules are presented to human experts for validation and for detecting access control issues. We have evaluated our approach on two applications; one is open source while the other is a proprietary system built by our industry partner. The obtained results are very promising in terms of the quality of inferred rules and the access control vulnerabilities it helped detect.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging; D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement

Keywords

Access Control Policies, Inference, Machine Learning

1. INTRODUCTION

Broken access control is a widely recognised security issue in web applications; it leads to unauthorised accesses to sensitive data and system resources. The consequences can be dramatic, from information leakage to business services being shut down. According to a recent report by OWASP[1], broken access control is involved in three out of the top ten vulnerabilities: *A7 – Missing Function Level Access Control*, *A4 – Insecure Direct Object References*, and *A2 – Broken Authentication and Session Management* [9]. Due to a lack of proper access enforcement, many web applications check access rights before making functionality or system resources visible (via web links, for example). However, the same check must be carried out on the server side when a functionality or a resource is accessed. Otherwise, attackers can forge requests to get access to the resource without being authorised (A7). The second vulnerability (A4) refers to the exposure of direct references to internal resources (such as files). If such internal resources are not adequately protected, they can be maliciously accessed. The third vulnerability (A2) relates to the authentication and session management of an access control mechanism. If they are implemented improperly, attackers can compromise user credentials or impersonate other users to access their private resources.

As web applications are integrating and providing more services to increasingly diverse entities (users, devices, or other service providers – which often have different functionalities and data), they are becoming complex. As a result, access control engineering for web applications becomes increasingly challenging. Broken access control is a likely risk if the access control model of a system is not designed and documented properly or the access control implementation is not adequately tested. For this reason, access control testing and validation is crucial to decrease the level of risk.

In general, access control (AC) consists of two artefacts: *AC policy specifications* and *AC mechanisms* that implement and enforce AC policies [10]. AC policies can be specified explicitly using models [4], such as RBAC (Role-Based Access Control) [21], or rules, such as XACML [18]. Such AC models must be correctly implemented and supported by runtime verification mechanisms. Moreover, regardless of the types of AC model or implementation, we have to ensure the coverage of AC policies and implementation. It is often the case that *not all resources that need access protection are properly covered by the policies and that such policies are not enforced by the implementation.* Another problem is that, in practice, many systems use hard-coded AC policies in their business logic code and do so without documentation. This implies that, more often than not, there is no AC policy specification available for testing and validation.

SACMAT'15, June1–3, 2015, Vienna, Austria.
Copyright © 2015 ACM 978-1-4503-3556-0/15/06 ...$15.00.
http://dx.doi.org/10.1145/2752952.2752969 .

[1]https://www.owasp.org

As a result, testing requires more human effort and its cost increases.

In this paper, we propose a semi-automated approach to the inference of AC policies for web applications. Taking advantage of a suite of security tools, our approach automatically exercises a target system and builds access spaces for a set of known users and roles. An access space of a specific user consists of discovered resources and the permissions the user has to these resources. Different abstraction techniques can be applied to the resources in order to group them into a higher level representation. Then, we are in a position to apply a machine learning to infer access rules for resources , at the chosen level of abstraction and roles. Certain rules may appear to be "Inconsistent", that is not consistently granting access to a resource for different users having a given role. They are then subject to further testing and analysis for refinement. Finally, the inferred rules can be validated against the specification of policies if they exist. Otherwise, they are presented to human experts for validation. The output rules help pinpoint AC problems, including: (i) resources that are left unprotected by the AC implementation, (ii) mismatch between the actual AC enforcement and the expected AC policies. Also, such rules, once validated, might be used in regression AC testing of future releases of web applications to detect regression faults in access control.

We have carried out an experimental evaluation of the proposed approach on two web applications, iTrust– an open source prototype for electronic health care management, and ISP– a crisis management system developed by our industry partner. The obtained results show that our approach is effective in discovering resources, determining access permissions, and inferring AC policies as expected. Furthermore, such policies help detect many AC problems including AC vulnerabilities that might lead to privilege escalation attacks and resources that are unprotected by AC implementation.

The remainder of the paper is organised as follows. Section 2 provides background concepts and discusses related work. Section 3 introduces a motivating example, followed by Section 4 where we discuss in detail the proposed approach. Section 5 shows the evaluation of the approach and analyses the outcome. Finally, Section 6 concludes the paper and outlines our ongoing work.

2. BACKGROUND AND RELATED WORK

2.1 Background

Access Control is a pervasive security mechanism which is used in virtually all systems [7]. It is concerned with authorising the right resource access permissions to users. The fundamental concepts in an access control model include *users*, *subjects*, *objects*, and *permissions*. *User* refers to human users who interact with a computer system. *Subject* refers to a process or program acting on behalf of a user. *Object* refers to resources accessible from a system, *Object* and *resource* are used interchangeably. *Permission* refers to the authorisation to perform some actions (e.g., read, update) on objects. In some systems, the notion of *Access Context* is also important, it concerns properties of subjects (e.g., location, age) who access, states of objects being accessed (e.g., a *paper* cannot be modified after the proceeding has been published), contextual factors when the access is taking place (e.g., working time or holiday), and access methods (e.g, using a trusted device or not). Over the past decade, a number of access control models have been proposed. Among them, the Role-based Access Control (RBAC) model [8] is the most widely adopted. *Role* is the most important concept in RBAC and refers to different privileges on a system. Users and permissions are assigned to roles; these assignments govern users' accesses to resources.

Web Crawling, also referred to as *Web Spidering*, is a technique for the retrieval of web resources. Its basic process is rather straightforward, starting from one or a few seeding (entry) pages, a web crawler (also called spider) extracts hyperlinks from the contents of the pages and then iteratively navigates the web pages addressed by those links [19]. Advanced web crawlers can identify and submit web forms to achieve more thorough exploration of web applications. Recent advances in web technologies (i.e., Web 2.0), where Javascript is used extensively to render user interfaces, implement navigation, and to enable asynchronous communications with web servers, have brought great challenges to web crawling as discussed in [23]. Unfortunately, only limited research attention has been paid to improve web crawlers to support Web 2.0, e.g., [16, 6].

Web crawling is used for a variety of purposes, such as web archiving, building search engines [19], or reverse engineering of web applications where models of web systems are reconstructed and used for maintenance or testing [5]. In security contexts, it helps scanning the "attack surface" of web applications, through which security test inputs, i.e., attacks, can be submitted to applications.

2.2 Related Work

Our work belongs to the research family of applying dynamic analysis to infer program specifications for detecting security vulnerabilities. In the area of access control analysis, a technique called "differential analysis" has been proposed to detect authorisation vulnerabilities in web applications [1]. It involves crawling a system under test using different authenticated users' sessions and unauthenticated ones in order to determine which portions of the system are accessible from which users. Our approach also involves web crawling with a set of user credentials, but we improve such a technique to account for Javascript and "unlinked" areas of web applications. Moreover, the goal of our approach is different, we aim at recovering AC policies for web applications. We consider different resource abstractions and apply machine learning to infer AC policies for roles, not users. As a result, our approach is more scalable to deal with complex web applications.

Noseevich et al. extended the differential analysis with the role concept and the notion of user cases, which represents roles' actions and their dependencies [17]. These are inputs defined by a human operator. The approach, then, considers sequences of use cases, iterates through these sequences and applies differential analysis for each user case in order to detect insufficient access control. Our approach differs in a number of aspects. First, our goal is to infer AC policies that, on the one hand, can be validated to detect AC issues. On the other hand, such inferred AC policies can be used for other purposes, e.g., regression testing or software maintenance. Second, we deal with Javascript and unlinked resources, which are omitted in the related work.

Alalfi et al. have conducted a number of work on the reverse engineering of RBAC models for web applications.

Source code transformation and instrumentation techniques and tools were proposed to recover UML-based structural models and behavioural models [2]. Such models act as inputs for another model transformation technique to construct RBAC models that can be used to check against security properties [3]. Our approach uses a proxy to obtain access traces, hence, we do not need to modify application source code. In addition, we take into account all types of server resources, including static files (e.g., PDF documents, images), which are not considered in Alalfi's approach.

On a larger scope, Slankas et al. [22] proposed an approach to extract access control policies from natural language text. Natural language processing techniques are used to extract access control concepts like subjects, actions, and resources. One of our applications used for evaluation is also used in [22]. Xiao et al. [25] proposed an approach, called Text2Policy, to extract automatically AC policies in XACML format from software documents and resource access information written in natural language.

Outputs of our approach are Role-based AC policies inferred along with execution logs that contain concrete web requests and responses. Our subsequent work will investigate an approach to transform them into models used for test case generation. Existing work on model-based test generation for access control testing, e.g., [12, 20, 13, 26], can be applied or extended towards the goal of having a complete and effective automated test generation for AC.

3. MOTIVATING EXAMPLE

We consider a simple web-based document management (SDM) system, depicted in Figure 1, as a motivating example for our approach. Its server-side code consists of five server pages: *LogIn*, *Main*, *viewDocument*, *manageDocument*, and *manageUser*; they are located in the following folders: *root*, *root*, *root/user*, *root/manager*, *root/admin*, respectively. The folder *root* is the root server directory of SDM. When a user accesses to the system, she is sent to the *Login* page and, once authenticated, she is redirected to the *Main* page in which she will be provided links to the *viewDocument* and *manageDocument* pages, depending on her roles in the system. *viewDocument* has two parameters, *path* and *id*, which indicate the path to a file and document identifier, respectively.

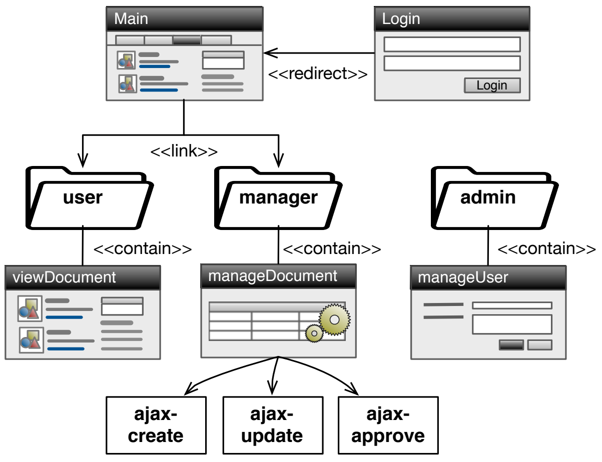

Figure 1: A simple web-based document management system.

In particular, the page *manageDocument* has three Ajax-based functions, namely *ajax-create*, *ajax-update*, and *ajax-approve* to create, update, and approve documents. These functions are also enabled based on users' roles and, once triggered, they open dynamic forms through which documents can be uploaded and updated.

The administrator of the system has to enter the *admin* suffix manually to the URL of SDM in order to access the administrative zone. The page *manageUser* allows him to create, update, delete, and assign roles to users. There is no direct link to the admin area from the *Main* page.

SDM adopts a role-based access control model to restrict accesses to the functionality of the system. Five roles are pre-defined: *adminRole*, *userRole*, *managerRole*, *authorRole*, and *guestRole*. Access permissions are assigned to roles as follows:

Role	**can access**
userRole	viewDocument
managerRole	manageDocument/ajax-approve
authorRole	manageDocument/ajax-create
	manageDocument/ajax-update
adminRole	manageUser

By design, all users can access the *Login* page while only authenticated users can access the *Main* page. From there, depending on the roles of a user, authorised functions (i.e., links) will be provided. Moreover, users of a given role *userRole* can only view their documents. Let us discuss some of the commonly encountered issues in web applications using SDM as an example:

Wrong Assumption: Once logged in, a user is redirected to the *Main* page. In this page, only links to the allowed pages are displayed. For example, assuming a malicious user named Mallory has been assigned to *userRole*, only a list of links of the form *viewDocument?path=...* are displayed on his browser. The developer makes an assumption that by displaying only authorised links a user like Mallory cannot access other pages. However, this assumption is wrong because based on indexing and suggesting tools like Google, by looking at a browser's history, or by simple guessing, Mallory can easily access other web pages if proper access enforcement is not in place. A malicious user can also try to guess and add some commonly used nouns like *admin*, *backup*, *config* to the main URL, in order to access sensitive areas of a web application. In the case of SDM, if only the administrator is assumed to know the *admin* suffix and no access control is in place, then unauthorised access is likely to occur.

Missing Protection: Since web clients can send direct HTTP requests to SDM, a malicious user can create requests to trigger the Ajax-based functions. If accesses are enforced only at the web pages, unauthorised accesses to these functions will occur.

Path Traversal: The *path* parameter of the *viewDocument* page can easily be tampered with illegitimate values. For example, a malicious user can enter a value such as */etc/passwd*, trying to compromise the server. Based on the naming scheme of the parameter an attacker can guess and access other users' documents. For example, if the values of the path parameter for Mallory are *mallory/1* and *mallory/2*, the pattern cam be inferred and Mallory can try the *alice/[0-9]+* pattern in order to view the documents of the user *alice*.

In the next section, we discuss our approach and how it aims at discovering resources in web applications like SDM, learning access permissions for different roles in an automated fashion, and pointing out AC issues.

4. APPROACH

Our approach, as illustrated in Figure 2, consists of five main steps: (1) *automated exploratory access testing*, (2) *resource access analysis*, (3) *inferring access rules*, (4) *rule assessment*, and (5) *targeted incremental access testing*. Our approach takes as input a set of credentials of different users who belong to different roles in a system under test (SUT). In the first step, we leverage a suite of security tools to explore (crawl) the SUT dynamically in order to generate access logs (HTTP requests/responses) for each user. During the second step, we perform resource extraction from the access logs and determine access permissions for users. In the third step, based on data about users' accesses obtained from the second step, we use machine learning to infer a decision tree that characterises access control policies. Step 4 assesses the decision tree to identify inconsistent rules. Step 5 performs additional access tests for the inconsistent rules that require more training data and refinement before iterating to the second step. Finally, we report the inferred rules to a human expert for validation and identification of AC problems. In the subsequent sections, we discuss each step in detail.

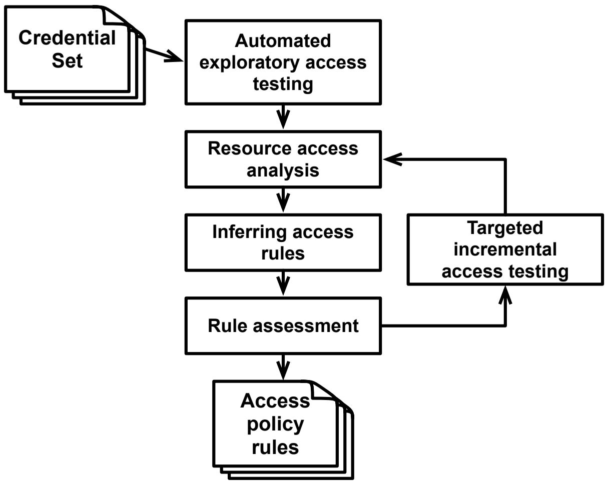

Figure 2: An overview of the proposed approach.

4.1 Automated Exploratory Access Testing

The goal of this step is to determine user access space for a given set of users. Depending on the assigned roles and roles' permissions, each user is *allowed* to access to some resources and is *denied* to access to the other unauthorised resources. We define the user access space of the SUT as a tuple $AS = \{U, G, R, C, p\}$, where U is a set of users; G is a set of roles; R set of resources of the SUT; C is a set of access contexts; and finally p is *allowed* or *denied*.

Figure 3 depicts our approach and the tools that we use for exploring a SUT and generate resource access logs. In our approach, we make use of a security tool suite called *BurpSuite*[2] to determine user access space. The BurpSuite's spider is used to discover the SUT's resources the users can access. It takes users' credentials and automatically analyses web contents for forms and links, submits the forms or follows the links to further explore the SUT. The chosen spider shows good crawling capability in most of the applications we used for evaluation. However, it needs support to deal with Javascript codes so that the dynamically rendered contents (e.g., menu links created with Javascript) can be explored. This is important because modern web applications use Javascript and asynchronous communication extensively to update the content, including links and forms, of web pages. In our motivating example, the three Ajax-based resources can only be discovered if Javascript is executed.

In addition, a chain of automated security tools (other spiders with different crawling capabilities or web scanners like Nikto2[3]) or testers, using a browser with Javascript enabled, can interact with the SUT. The SUT's discovery results, including HTTP requests and responses, are captured by BurpSuite's proxy.

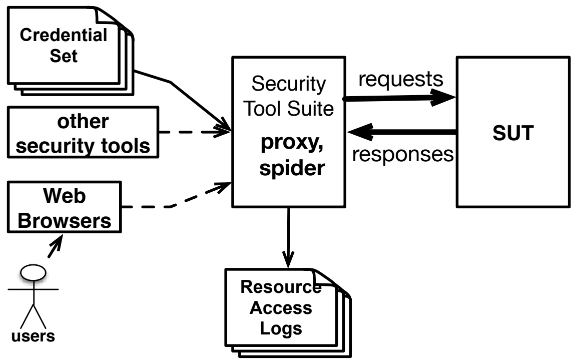

Figure 3: Automated exploration of resource accesses for a SUT.

Taking the intercepted responses as entry points and user's provided data in their requests, the spidering tool can explore the SUT more thoroughly for hidden resources that are unlinked or are accessible only with specific data and/or by following some business flows. For example, considering our motivating example, since the administration pages are isolated from the main page, the spider alone may not detect them. However, if an entry point to those pages is provided by a user and captured by the proxy, the spider can use the entry point to explore the pages for resources. Moreover, when the SUT has Web test cases (such as Selenium[4] tests), which access the SUT via Web interface, we can execute such tests over the proxy to provide more entry points and valid data to the spider. This will help the spider quickly uncover more resources.

It is worth noticing that there are other commercial or open source security tool suites that provide similar features, such as WebInspect[5] and Acunetix WVS[6]. Since our approach analyses the standard HTTP requests and responses, it can be easily integrated with those tools as well. We selected BurpSuite because of it has shown good performance and is more affordable compared to other commercial tools. Besides, it also has a free community version for limited adoption.

Each user is considered separately in our exploration approach. Her credential is fed to the security tool suite, which

[2]http://portswigger.net/burp

[3]https://cirt.net/Nikto2

[4]http://www.seleniumhq.org

[5]https://download.hpsmartupdate.com/webinspect/

[6]http://www.acunetix.com/vulnerability-scanner/

will then automatically crawl the SUT until no new resource access is discovered. During the session, the developer can optionally access the SUT so as to provide meaningful input data and more entry points to aid the crawling process. Once all users have been considered, we combine all the discovered resource URLs and their corresponding requests to perform user-resource access testing for each user and all the resources that were not discovered within her exploratory session. The goal is to check the access permissions of each user on all resources.

Moreover, we need to provide the crawling tool with one or more entry (i.e., seeding) pages so that the crawling process can start from them. For the SUTs that do not provide access to their server directories, we can use the usual spidering method that is provided with the home/login page as the only entry point. We name this method *single-entry* exploration.

For SUTs that have their server directories accessible for analysis, we propose an exploration method called *all-entry*, in which we consider all server static resources, including server pages, images, folders, and all other files located in the server directory, as entry points for spidering. The goal is to improve the crawler's performance by providing it with more seeding pages. In fact, for *all-entry*, we can easily scan the server directory for all static resources (e.g., all JSP pages using the Linux *find* command) and issue HTTP requests to access them. Such requests and their responses are captured by the proxy and served as entry points for the spider.

The outputs of this phase are *resource access logs* that contain detailed data about all the issued HTTP requests (resource URLs, parameters, etc.) and server HTTP responses. The next phase focuses on analysing these logs for determining meaningful access spaces.

4.2 Resource Access Analysis

Resource access logs contain session identifiers, HTTP requests, and HTTP responses. From them, we need to extract important information for access control analysis, including *user*, *target resource*, and *access permission*. Since in the previous phase each user has a separate exploratory session, mapping the log entries to users is straightforward. Hence, this section discusses in detail two tasks: (1) extracting resources, and (2) determining access permissions on the identified resources.

4.2.1 Extracting Resources

We propose to extract resources that need to be considered in access control from the logged HTTP requests, especially from the URIs (Uniform Resource Identifiers) of the GET requests and from the URIs combined with the message body contents of the POST requests [24]. The URI of a GET request is composed of a path to a server resource and a query, which is specified as pairs of parameters and their values. The URIs of POST requests often contain only paths to server pages; the content of the request is also specified as pairs of parameters and their values. We name the path section of a URI of any HTTP request as *base URI*. In general, a base URI indicates a web element, for example, a server page/servlet/CGI script, a directory, or a file. They are often resources that need to be protected. However, in addition to the base URIs, some applications also use request parameters to identify resources and, in such cases, relevant parameters and concrete values must be taken into account in extracting resources.

We propose two levels of resource identification: *base-URI-resource*, and *full-resource*. At the base-URI-resource level, we consider the base URIs as resources, stripping out all parameters. At the full-resource level, we are interested in controlling access to specific resources that are identified based on request parameters. Therefore, we have to consider also the parameters and their values as provided in the requests that help identifying resources. For example, the parameter *id* of the page *viewDocument* can have multiple integer values in the access logs, each identifying a specific document that can only be accessed by the owner of the document and users belonging to the *managerRole* role. The challenge at this level is to know which parameters are relevant for determining resources. For example, the requests that point to the URI *manageDocument/ajax-update* could have a number of parameters such as *id*, *title*, *date*, where only the first one is relevant for resource identification. In this work, we need to involve the developer to provide a list of parameters that can have an impact on access control policies. Future work will look into automated solutions to identify such parameters.

By default, our approach considers the base-URI-resource level. Since our approach is iterative, in subsequent steps, if we find inconsistent AC rules for some resources because of missing parameters, we will go back to this step and consider the full-resource level for them. Base-URI resources will be decomposed into more concrete resources, taking into account relevant parameters and values. The output of this extraction task is the set of identified resources R.

4.2.2 Determining Access Permission

This task focuses on determining access contexts C and granted permissions P for the identified resources. We analyse the information available in the access logs, specifically the HTTP requests and responses, to decide in which context a resource is requested, and the permissions the SUT grants to the user who is accessing the resource.

Various factors can contribute to determining access contexts, such as *resource states*, *access methods*, and *access history*. Each resource may have its own set of states, identifying them and determining their roles in access control automatically is non-trivial. As a result, we reserve resource states for future work. Access methods are the standard HTTP request methods [24], including *GET*, *POST*, and *DELETE*. Access history is a list of requests prior to the current resource accesses and reflects the sequence of activities characterising how a user interacted with the SUT. Such a sequence of activities is often governed by the business process model of the SUT and might affect whether a request is allowed or denied. For example, a *payment* request is allowed only if some items were previously added to a shopping cart.

In this work, access context includes access methods and the *referrer* attribute of HTTP requests that indicates the prior page that links to the current resource being requested. This attribute, to a limited extent, captures access history. Future work will investigate more complex definitions of context.

Access permission of a user to a resource is determined based on HTTP responses, more specifically the standard HTTP status codes [24] and the HTML contents. In addition to using HTTP status codes, some web systems gracefully deny unauthorised accesses with an "OK Status" web page (HTTP code = 200) stating "Access is not allowed". These denial messages usually follow one or a few specific patterns and we, therefore, attempt to use regular expressions to match HTML contents and classify them into *denial* or *normal*. The former indicates that its corresponding request was denied while the latter implies otherwise. Following are the rules to determine access permission:

Condition	Permission
HTTP Code = 4xx,5xx,301, any content	denied
HTTP Code = 200,302,304, *denial* content	denied
HTTP Code = 200,302,304, *normal* content	allowed

4.3 Inferring Access Rules

After resource access logs have been analysed from the previous steps we obtain a data corpus characterising the access space $AS = \{U, G, R, C, p\}$. It can be presented as a table with the following columns (attributes): user, role, resource, context-method (i.e., HTTP access methods), context-referrer (i.e., referrers in HTTP requests), and permission.

Since the number of resources is often large, their combinations with other attributes tend to yield large tables preventing manual analysis. We propose to apply a classification technique called *decision tree*, from machine learning, which is widely used in many fields to infer rules. The goal is to aid the analyst to infer quickly access rules from the data collected, check the rules against expectations, and identify potential issues and inconsistencies.

We employ a machine classifier implemented in Weka[7], a popular machine-learning tool suite. Specifically, we use the *RandomTree* classifier because its output decision trees are interpretable. Furthermore, in our preliminary study about alternative classification methods, we have found that *RandomTree* yields good classification precision and recall when processing our data. Attribute *permission*, which takes the value *allowed* or *denied*, acts as the class (or label) in the data samples. A learned decision tree can be graphically represented as a tree (e.g., Figure 4) in which the root and intermediate nodes are the predictor attributes, outgoing edges from the nodes are specific values for these attributes, and each leaf node represents a predicted class label (allowed or denied in this context). For example, in Figure 4, the root node is the resource attribute and if resource is r_1 then the next decision node is the role attribute. If role is $role_1$ and then the next node is the context attribute. Finally, if context is c_1 then the access is allowed.

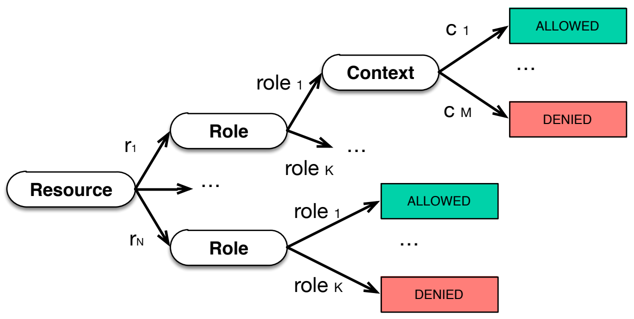

Figure 4: Graphical representation of decision trees learned to infer access rules.

More interestingly, the paths from the root to the leaves represent classification rules. In our context, they are the AC policy rules that we need to infer. It is important to note that the user attribute is not in the tree and rules since, in most systems (e.g., those that use the RBAC model), access permissions are assigned to roles. Therefore, the user attribute does not participate in decision-making and is not accounted for by the classifier. However, in some cases where the concept of ownership is relevant, i.e., only owners of some resources can access them, the user attribute may be accounted for and appear in the decision tree and rules. Following example shows three concrete rules that we inferred from the experimental systems. Rule 1 states that everyone is allowed to access the 'login.jsp' page. Rule 2 states that users of the role 'guest' cannot access the 'admin.jsp' page. Rule 3 states that users of role *author* can edit documents only when their status is 'draft'.

```
RULE 1:
     resource = 'login.jsp' : allowed
RULE 2:
     resource = 'admin.jsp'
     |   role = 'guest'
     |   |   method = 'GET' : denied
RULE 3:
     resource = 'edit.jsp'
     |   role = 'author'
     |   |   docStatus = 'draft' : allowed
     |   |   docStatus = 'post' : denied
```

The next section discusses how we assess the inferred rules, improve them if necessary, and finally help the analyst to validate the rules.

4.4 Rule Assessment

Apart from the learned decision tree and rules, the classifier provides the *prediction confidence* at each leaf node of the tree. This measure indicates the homogeneity of the data sample that forms the leaf node, in terms of percentages of accesses that are allowed or denied. We define *consistent* rules as those that predict allowed or denied with 100% confidence level, whereas the other rules are referred to as *inconsistent*, meaning that in their corresponding leaf nodes corresponding some accesses are allowed while others are denied. In the context of access control, such inconsistent rules may indicate potential problems in the access control implementation, or alternatively a lack of information regarding the attempted accesses (e.g., history, parameters), as further discussed below.

One possible reason for inconsistent rules is the abstraction process when extracting resources, where details of parameters are abstracted away to help scale the analysis. As a result, some resources considered in our analysis may actually represent groups of concrete resources for which different users may have different access permissions, even if they have the same role. This is due to some systems using request parameters to specify access operations (e.g., read, update, delete) or relying on the concept of ownership (e.g., users can edit the resources they create) or user profile (e.g., customers being 18 or older can purchase wine) to implement access control.

To deal with inconsistency we employ a step called *targeted incremental access testing* (see Figure 2). In this phase, we first select inconsistent rules and identify resources and roles that are involved in these rules. Identified resources and roles will help trace back to their original users, requests, and responses. The full-resource level of resource extraction should then be considered for such resources. Let

[7]http://www.cs.waikato.ac.nz/ ml/weka

us consider our motivating example. If we consider the URI *user/viewDocument* as a target resource, different users of the role *userRole* can hold different access permissions since, during the exploratory phase, they access different documents identified by the id parameter. Therefore, for *user/viewDocument* we need to consider its full-resource level, i.e., including the id parameter and its values. As a result, instead of having *user/viewDocument* as the only resource, we have to consider *user/viewDocument?id=1*, *user/viewDocument?id=2*, etc., assuming 1 and 2 are two example values of id.

In the step *Targeted incremental access testing*, we currently involve the developer to identify relevant parameters and their input values for the resources involved in the inconsistent rules. Then, requests accessing these resources are generated by combining their base URI and those parameter values. A t-way test combinatorial technique [11] can be applied to reduce the number of combinations to be exercised in such new requests. The generated requests are submitted to the SUT for every user (among those selected), and then the obtained access logs are analysed again to update the access space and the inferred policy rules. This iterative process is repeated until inconsistent rules are resolved. Finally, the obtained policy rules are validated by the developer.

All the inferred rules should be checked to pinpoint AC problems and discrepancies with expected AC requirements. However, since their number could be large, we suggest the developer should, first, pay special attention to the following types of resources and their corresponding access rules:

- Resources that allow access to all users; we have observed many cases where such resources have been left unprotected by the implemented AC mechanism.
- Resources that are related to database, configuration, installation, backup files, and other static documents. The development team might have forgotten to clean them up or properly protect them before deployment, and there could potentially be insecure direct accesses (without authentication) to them.
- Resources with corresponding inconsistent AC rules. In practice, AC policies might be governed by other factors, such as subject profiles or resource states. We may have to limit incremental access testing iterations because of time constraints, hence not fully resolving all inconsistent rules. In such cases, the developer should check whether inconsistency is due to AC implementation mistakes or other reasons that affect access control.

5. EXPERIMENTS AND RESULTS

In this section, we discuss the experiments that were carried out to evaluate our approach. We investigate whether the proposed approach is effective in discovering resources and inferring correct AC rules. We also assess whether such rules are helping to detect AC vulnerabilities. More specifically, we aim at addressing the following research questions:

RQ1: *Does the proposed approach effectively discover resources for inferring AC rules?*

RQ2: *What is the quality in terms of correctness and consistency of the inferred AC rules?*

RQ3: *How useful are the inferred AC rules in detecting AC issues?*

In what follows, we present the applications selected for the experiments, the experimental procedure, and the results we obtained.

5.1 Applications

Our criteria to select web-based systems for our evaluation are twofold. First, they must implement an access control mechanism that supports the basic concepts of *user*, *role*, *resource*, and *permissions* of roles to access resources. Second, we must know beforehand the users, roles, and user-role assignments in selected systems, which is a pre-requisite to be able to learn access control rules.

We have selected two systems, ISP and iTrust, for our experimental evaluation. ISP is a complex crisis management system under development by an industry partner involved in this work. (The full name of the system and the name of the company are omitted because of the anonymity requirement of the conference.) ISP is used in disaster relief and humanitarian missions to manage and process sensitive data. It involves mobile teams that are deployed on the field to collect data and synchronise them with crisis management centres via satellite communications. In the system, users are assigned to different roles (for example, *ground-team*, *mission-lead*) that have different access permissions to resources, e.g., missions. ISP implements an AC administration module where users and roles can be defined at run-time. ISP distinguishes 49 groups of resources, called *logical* resources so that administrators of the system can efficiently assign access permissions to roles. In addition, from a technology standpoint, Javascript is extensively used in ISP to render the user interface and for Ajax communications with the web server.

iTrust is a feature-rich prototype and open source system for electronic health care management [15]. It has been developed at North Carolina State University to provide software engineering students with an educational project that is relevant and has enough depth and complexity. Moreover, it also provides an educational testbed for understanding the importance of security and privacy requirements. iTrust has been used in previous studies related to access control [14, 22].

iTrust predefines eight roles including *doctor*, *patient*, and *staff*. The definition of roles and role access permissions are carried out at design time. Authorised users of a role can access resources in specific folders. For example, the users with role *patient* are allowed to access resource files under the */auth/patient/* and */auth/* folders. These access control rules are specified in the configuration file (WEB-INF/web.xml) of iTrust and enforced by the web container where iTrust is deployed, e.g., apache-tomcat[8]. We use this configuration as a baseline of comparison ("gold standard") when evaluating inferred access rules for iTrust.

5.2 Tool and Experiment

We have developed a tool that works in tandem with BurpSuite to support our approach. The tool takes as inputs configuration files that specify user credentials and content patterns, which are used in the determination of access permissions as discussed in Section 4.2. It interacts with BurpSuite to feed login credentials, to start spidering sessions, and to send requests to web servers through BurpSuite's proxy so that requests and responses can be captured and

[8]http://tomcat.apache.org

used for analysis. Once the spidering process is finished, the tool can analyse the log data of BurpSuite for identifying resources and determining access permissions. This tool is available for download[9].

Since ISP needs roles to be defined at run-time (except the most privileged *admin* role), we created five roles having different sets of permissions on resources. Then, for each application we prepared a set of users and assigned them to roles: 15 users for iTrust, and 14 users for ISP. As for correctly classifying access permissions, we defined five content patterns for iTrust. For example, the pattern "*your role is invalid*" is used to classify accesses as denied if it is contained in their corresponding responses. Similarly, we defined ten additional content patterns for ISP. The definition of such patterns is straightforward; we can try some unauthorised accesses to learn the response patterns or we can simply ask the developer for the coding convention of denial messages.

Users' credentials (*username* and *password*) and content patterns are fed into our tool, which exercises the web applications and return the user access space, i.e., the data corpus ready for applying machine learning. Finally, the developer can assess the inferred AC rules, following the guidelines discussed in the rule assessment step of our approach, and perform additional access testing to refine inconsistent rules where needed.

To answer the research questions, we rely on the following experimental data: (1) the number and percentage of discovered resources in comparison with static resources when applicable, (2) the correctness and completeness of inferred AC rules when compared to the gold standard, and (3) the number of AC issues detected and their cause.

5.3 Results

In this section, we present in turn the experimental results obtained from iTrust and ISP.

5.3.1 iTrust

Regarding iTrust, we used the base-URI-resource extraction level, and consequently no request parameters were considered. When applying the single-entry exploration technique, only the home page of iTrust was fed into our framework. As a result, it was able to discover 130 out of a total of 248 resources. When all-entry exploration was performed, all 248 resources were detected. However, among them, 44 resources returned Java exception error pages because of implementation errors or due to the fact they were not intended to be accessed directly but only through other pages. For example, *header.jsp* or *footer.jsp* are included in all other web pages. In terms of access control, these 44 resources are considered to be vulnerabilities (of the type *A4 – Insecure Direct Object References*) since their direct access are allowed to all users, and even worse, source code and database information is disclosed in the error pages. We exclude them from the next step, AC rule inference, as it is clear that they are accessible to all users. Since the results of all-entry subsumes the results of single-entry in terms of resources, inferred AC rules, as well as issues found, in the following we analyse the results of all-entry only.

Based on the 207 discovered resources that remain for further analysis, our technique has inferred 1518 AC rules for resources, roles, and permissions. Among them, 1441 are correct with respect to the gold standard AC rules of iTrust.

[9]URL removed for anonymity.

These inferred rules encompass all the gold standard rules thus reaching the maximum completeness of 100%. The remaining 77 inferred AC rules for 43 resources need further validation. Analysing them we found that:

- 38 resources are uncovered by the implemented AC rules (*A7 – Missing Function Level Access Control*). In other words, they are not checked for authorisation. Most of them are located in the *util*, *errors*, and *DataTables* folders. These resources can leak important information, such as showing database tables or transactional logs, or can lead to privilege escalation. We realised that only some of them are intended for end users, while the others are for development purposes only. This is the reason why there are so many resources that are left unprotected.

- Five resources are false positives. Three of them are intended to be accessible without authentication (*/privacyPolicy.jsp*, */j_security_check*, */css.jsp*), while the other two are system files that should be ignored by our tool as their access is controlled by the web server (*/WEB-INF/web.xml*, and */META-INF/context.xml*).

Among the unprotected resources for end users, we found three privilege vulnerabilities: (1) */util/resetPassword.jsp* allows any user (even a guest without authentication) to change passwords. It has a simple question-answering scheme, and no old password is required. As a result, it is very easy to change the passwords of other users, including the admin user, to gain access to the system. (2) The */util/getUser.jsp* page is designed to be invoked from other pages where the developer assumed that authorisation has already been enforced. However, it is also accessible independently and does not re-enforce authorisation in such a case. As a result, any user can query private data from other users. (3) The */errors/reboot.jsp* page allows anyone to reboot the web server, which might render the system inaccessible to all users. The first and second vulnerability can be categorised as *A2 – Broken Authentication and Session Management*, while the third one belongs to *A7 – Missing Function Level Access Control*.

Among the 1441 correctly-inferred AC rules, 11 AC rules (0.8%) for seven resources are inconsistent. We have performed one iteration of incremental access testing to resolve the inconsistencies by refining relevant branches of the decision tree. No parameter of these seven resources is relevant to determine more resources or access permissions. As a result, we applied our *Resource Access Analysis* step, refining a content filter, to correct the access permissions of the requests to six (out of seven) resources, which were classified as "denied" due to an application error (*index out of bound*), while the correct permission should have been "allowed". The AC rules of the last resource, */auth/patient/addTelemedicineData.jsp*, remains inconsistent:

```
RULE:
      resource = patient/addTelemedicineData.jsp
      |   role = Patient
      |   |   method = GET : denied (75.0
      |   |   method = POST : allowed (62.5
```

Investigating this resource we found that there is no relevant parameter that participates in AC. Also, according to the gold standard, all requests from the role *Patient* should be allowed, which is not the case in this rule where 75% GET and 37.5% POST requests are denied. The reason is that iTrust implements an additional AC policy that controls access based on patient profiles: access permission is only allowed to patients that have a specific profile regarding their *blood pressure, glucose level, height, and weight.* To make this rule consistent, we would have to take into account different classes of patients according to such a profile.

Looking more carefully at the rule, we find that the permissions determined for POST and GET are different. This indicates a potential flaw in the implementation. We looked further into the source code of the resource and confirmed that this is indeed an AC vulnerability. The AC check based on patient profiles is missing for POST requests, and as a consequence, many direct POST requests were allowed even if they came from patients that did not have the required profile.

In summary, in the evaluation on iTrust, our approach discovered 130 and 248 resources (i.e., 100%), respectively, with single-entry and all-entry exploration. All-entry is provided with more seeding pages for crawling. As a result, it is more effective than single-entry. Our approach inferred 1518 AC rules in which 1441 (94.92%) are correct compared to the gold standard. The remaining 5.08% need further inspections since their corresponding resources are not protected by the AC implementation. Our approach has also indicated many AC issues in iTrust: 44 resources show compilation errors revealing source code and valuable information for malicious exploitation; 38 resources are uncovered by access control implementation leading to three AC vulnerabilities; one resource (pertaining to an inconsistent AC rule) is improperly protected given that access is driven by patient profiles. This resource is vulnerable to privilege escalation attacks since a patient could change other patients' data.

5.3.2 ISP

Regarding ISP, we could only apply the single-entry exploration technique since we had limited access to the system at the time of running the evaluation. However, thanks to the support of the proxy, we could also use web browsers during the crawling process to access ISP so that interfaces were rendered and Javascript-based navigation was used by the crawler. This helped discover more resources.

We compared two runs of the exploratory step: one is completely automated by BurpSuite (i.e., no Javascript is executed) while the other is enhanced with Javascript execution. For the latter, we had involved one developer using a web browser to traverse ISP pages in a depth-first manner, through the exploratory session of an admin user (who can access to all resources). It took him four working hours to complete the task, which is reasonable compared to the effort that can be saved using the technique. The run without Javascript support returned 353 resources. The run with Javascript support discovered 680 resources, which is significantly higher than the number of resources discovered when Javascript execution is ignored. This shows the importance of considering Javascript, which our approach does, in resource discovery.

After excluding Javascript, CSS, and unimportant (with respect to information disclosure) image files from the discovered 680 resources, there were 399 resources left for analysis. We inferred and analysed AC rules for a subset of 131 randomly selected resources since their mapping to logical resources required substantial manual effort among system engineers. In what follows we discuss three interesting and representative situations.

- There were 15 CSV and JSON files that were unprotected from direct accesses (*A4 – Insecure Direct Object References*). Unauthorised users can access them if they know the URLs.

- Five resources have accesses that are specified by 15 inconsistent rules. Inspecting them, we detected a number of discrepancies between the defined AC rules, using the ISP's AC administration module, and the inferred AC rules. This was due to access control enforcement being different from the defined rules. For example, according to the defined AC rules, some users are allowed to create new data records while in reality their accesses to such a function are denied. ISP developers have confirmed these issues after discussion.

- 80 resources can be mapped to 16 logical resources based on their names. All 240 rules inferred for them are consistent. However, since the mapping has not been checked by the ISP developers due to time constraints, the correctness of the inferred AC rules is yet to be confirmed.

To support the analyst in inspecting inferred AC rules, our tool can filter rules of interest, including those that are inconsistent or allow/deny accesses from any user. We present them graphically as a decision tree in Freemind[10]. Figure 5 depicts an example of the graphical format of the inferred AC rules for ISP, where the analyst can expand/collapse and navigate the tree to check the inferred access permissions of roles to resources. Note that the leaves of the decision tree are labelled with the corresponding classification and confidence level.

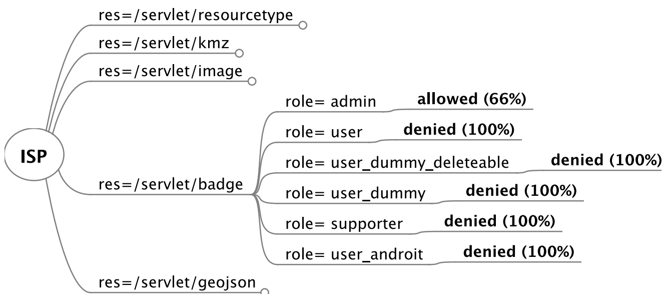

Figure 5: Examples of AC rules inferred for ISP.

5.4 Discussion

Based on the results presented above, we find that our approach is effective in discovering resources for inferring AC rules, especially with the all-entry exploration (**RQ1**). This technique discovered all iTrust resources considered in access control management. In the case of ISP, the single-entry exploration, augmented with the use of interactive web browsers for Javascript, has also resulted in a high number of resources being discovered.

RQ1: The proposed approach is effective in discovering resources when considering all-entry exploration and supporting Javascript.

The proposed technique yielded a high ratio of correctly inferred AC rules (**RQ2**): 94.92% (1441/1518) of the rules in the gold standard for iTrust. The remaining rules are not

[10]http://freemind.sourceforge.net/

necessarily incorrect since they were not specified. They pertain mostly to resources that are left unprotected by the AC mechanism in place. Among the correctly inferred AC rules, only about 0.8% (11/1441) were inconsistent and needed further refinement. The reason for inconsistency, as we have discussed earlier, is due to the misclassification of access permissions for six resources (due to incomplete content patterns) and an AC vulnerability for one resource. The former can be resolved by completing our list of content patterns and the latter requires to fix the AC implementation.

RQ2: The proposed approach can infer a high ratio of correct AC rules that highly, though not completely, encompass the gold standard.

Resources that are left unprotected by the AC implementation or have access permissions specified by inconsistent rules have a high probability of being affected by AC vulnerabilities. As the results we obtained for both applications have shown, unprotected resources can lead to leakage of valuable information or AC privilege escalation vulnerability. Apart from inferring AC rules, our technique can help detecting such problems as well (**RQ3**).

RQ3: Inconsistency in the inferred AC rules are helpful in detecting AC issues.

Though presenting many advantages, our approach is not fully automated yet. First, it involves manual effort in identifying parameters that are relevant for resource identification and inputs for incremental access testing. However, our approach is easier to apply when dealing with systems like iTrust in which access control is mainly done on the level of server files and folders, or systems that have a limited number of request parameters participating in AC. Second, the mapping between discovered resources and logical resources also requires manual effort. We are currently extending our work to address these limitations.

6. CONCLUSION

Broken access control is a widely recognised security issue in web applications. As web applications are becoming increasingly complex , access control (AC) vulnerabilities become significant threats. Testing and validation to detect AC problems are thus crucial but usually rely on predefined and regularly updated AC specifications. In practice, however, it is common for such specifications to be missing or outdated. For many systems, AC policies are hard-coded in the business logic code without proper documentation.

We propose in this paper a semi-automated, bottom-up approach to infer AC rules for web applications. The proposed approach rests on an integration of a spider, a web proxy, and web browsers for dealing with Javascript and user inputs. It first exercises a target system automatically and builds access spaces for a set of known users and roles. Then, machine learning is applied to infer AC rules. Problematic resources are identified if the accesses to them are specified by inconsistent or incorrect rules (checked by the developer) or they are left unprotected accidentally.

We have evaluated our approach and its supporting tool on two web applications, one open source and the other a proprietary system developed by our industrial partner. The results are very promising: our approach can effectively discover resources, determine and correctly infer access permissions and AC rules. It is also helpful in pinpointing many AC vulnerabilities, including insecure direct access to resources, privilege escalation, and faulty implementation of AC policies. In our ongoing work, we aim at addressing the limitations of this work with incremental testing and rule refinement. We will also investigate an approach to map actual resources to logical ones. These two extensions will help reducing the amount of manual effort that is needed with the current solution.

7. REFERENCES

[1] *Hacking Exposed Web Applications: Web Application Security Secrets and Solutions.* McGraw-Hill, 3rd edition, 2011.

[2] M. Alalfi, J. Cordy, and T. Dean. Automated reverse engineering of uml sequence diagrams for dynamic web applications. In *Software Testing, Verification and Validation Workshops, 2009. ICSTW '09. International Conference on*, pages 287–294, April 2009.

[3] M. Alalfi, J. Cordy, and T. Dean. Recovering role-based access control security models from dynamic web applications. In M. Brambilla, T. Tokuda, and R. Tolksdorf, editors, *Web Engineering*, volume 7387 of *Lecture Notes in Computer Science*, pages 121–136. Springer Berlin Heidelberg, 2012.

[4] N. Damianou, A. Bandara, M. Sloman, and E. Lupu. A survey of policy specification approaches. *Department of Computing, Imperial College of Science Technology and Medicine, London*, 2002.

[5] G. Di Lucca, M. Di Penta, G. Antoniol, and G. Casazza. An approach for reverse engineering of web-based applications. In *Reverse Engineering, 2001. Proceedings. Eighth Working Conference on*, pages 231–240, 2001.

[6] C. Duda, G. Frey, D. Kossmann, and C. Zhou. Ajaxsearch: Crawling, indexing and searching web 2.0 applications. *Proc. VLDB Endow.*, 1(2):1440–1443, Aug. 2008.

[7] D. Ferraiolo, D. R. Kuhn, and R. Chandramouli. *Role-based access control - 2nd edition.* Artech House, 2007.

[8] D. Ferraiolo and R. Kuhn. Role-based access control. In *In 15th NIST-NCSC National Computer Security Conference*, pages 554–563, 1992.

[9] T. O. Foundation. Owasp 10 most critical web application security risks. Technical report, OWASP, 2013.

[10] J. Hwang, E. Martin, T. Xie, and V. C. Hu. Testing access control policies. In *Encyclopedia of Software Engineering*, pages 673–683. 2010.

[11] R. Kuhn, R. Kacker, Y. Lei, and J. Hunter. Combinatorial software testing. *Computer*, 42(8):94 –96, aug. 2009.

[12] E. Martin. Automated test generation for access control policies. In *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*,

OOPSLA '06, pages 752–753, New York, NY, USA, 2006. ACM.

[13] A. Masood, R. Bhatti, A. Ghafoor, and A. P. Mathur. Scalable and effective test generation for role-based access control systems. *Software Engineering, IEEE Transactions on*, 35(5):654–668, 2009.

[14] A. K. Massey, P. N. Otto, L. J. Hayward, and A. I. Antón. Evaluating existing security and privacy requirements for legal compliance. *Requirements engineering*, 15(1):119–137, 2010.

[15] A. Meneely, B. Smith, and L. Williams. itrust electronic health care system: A case study.

[16] A. Mesbah, A. van Deursen, and S. Lenselink. Crawling Ajax-based web applications through dynamic analysis of user interface state changes. *ACM Transactions on the Web (TWEB)*, 6(1):3:1–3:30, 2012.

[17] G. Noseevich and A. Petukhov. Detecting insufficient access control in web applications. In *SysSec Workshop (SysSec), 2011 First*, pages 11–18, July 2011.

[18] OASIS. Extensible access control markup language (xacml). Technical report, OASIS, 2003.

[19] C. Olston and M. Najork. Web crawling. *Foundations and Trends in Information Retrieval*, 4(3):175–246, 2010.

[20] A. Pretschner, T. Mouelhi, and Y. Le Traon. Model-based tests for access control policies. In *Software Testing, Verification, and Validation, 2008 1st International Conference on*, pages 338–347. IEEE, 2008.

[21] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role-based access control models. *Computer*, 29(2):38–47, 1996.

[22] J. Slankas and L. Williams. Access control policy extraction from unconstrained natural language text. In *Social Computing (SocialCom), 2013 International Conference on*, pages 435–440, Sept 2013.

[23] P. Tonella and F. Ricca. Dynamic model extraction and statistical analysis of web applications: Follow-up after 6 years. In *Web Site Evolution, 2008. WSE 2008. 10th International Symposium on*, pages 3–10, Oct 2008.

[24] W3C. Hypertext transfer protocol – http/1.1, 1999.

[25] X. Xiao, A. Paradkar, S. Thummalapenta, and T. Xie. Automated extraction of security policies from natural-language software documents. In *ACM SIGSOFT FSE'12*, page 12. ACM, 2012.

[26] D. Xu, L. Thomas, M. Kent, T. Mouelhi, and Y. Le Traon. A model-based approach to automated testing of access control policies. In *Proceedings of the 17th ACM symposium on Access Control Models and Technologies*, pages 209–218. ACM, 2012.