# Detecting Insufficient Access Control in Web Applications

George Noseevich, Andrew Petukhov
Computer Systems Lab
Computer Science Dept.
Lomonosov Moscow State University
Moscow, Russia
Email: ngo@lvk.cs.msu.su, petand@lvk.cs.msu.su

***Abstract*—Web applications have become a de facto standard for delivering services on the internet. Often they contain sensitive data and provide functionality which should be protected from unauthorized access. Explicit access control policies can be leveraged for validating the access control, but, unfortunately, these policies are rarely defined in case of web applications. Previous research shows that access control flaws in web applications may be revealed with black-box analysis, but the existing "differential analysis" approach has certain limitations. We believe that taking the state of the web application into account could help to overcome the limitations of exiting approach. In this paper we propose a novel approach to black-box web application testing, which utilizes a use-case graph. The graph contains classes of actions within the web application and their dependencies. By traversing the graph and applying differential analysis at each step of the traversal, we were able to improve the accuracy of the method. The proposed method was implemented in the proof-of-concept tool AcCoRuTe. Evaluation with several real-world web applications demostrates better results in comparison to simple differential analysis.**

***Index Terms*—web applications; access control; vulnerability analysis; penetration testing;**

## I. Introduction

Web applications are an increasingly popular way of providing services and data on the internet. Unfortunately, they are often implemented by developers who lack security skills, or security is neglected due to time or financial constraints. This has resulted in an increase of web application vulnerabilities that were reported recently.

According to the statistics, the amount of traditional input validation errors such as SQL injections and XSS is decreasing[1]. Besides, automated techniques for detecting such vulnerabilities have improved significantly[2].

Automated detection of certain other vulnerability types, however, is not so advanced. For example, access control flaws typicaly require manual detection[3].

Authorization protects data and operations from unauthorized access. Whenever a user can successfully perform actions that should have been prohibited for him (e.g., transfer money from the account that does not belong to him), or access data that he is not allowed to view, there is an access control flaw.

One of the possible approaches for broken access control detection is leveraging explicit access control policy [3], [4]. Unfortunately, such a document is rarely available for existing web applications.

In the absence of an explicit access control policy we need a way of extracting it from the web application behavior. We have chosen to regard a web application as a black box and to use the application's interface to derive access control rules. Thus, we assume that users should only be authorized to perform actions that are listed in their web interface. We define an access control flaw as a possibility for certain user to successfuly perform an HTTP request that could not have originated from his web interface.

Several authors have addressed access control flaws by applying black-box web application analysis and leveraging the web interface as an implicit access control policy. Differential analysis ([5], [6]) suffers from several limitations, which reduce its effectiveness in large web applications with complex logic. The main reason for this is its inability to take the web application state (the state is determined by web application database contents, filesystem, etc) into account.

In this paper, we extend the existing method. By

[1]The Web Application Security Consortium statistics for 2008 [1] states that the number of web sites containing vulnerabilities of these two widespread types has decreased by 13% and 20% respectively as compared to 2007.

[2]According to WASC statistics, 29% of XSS vulnerabilities and 46% of SQL injection vulnerabilities were detected by automatic scanning with default settings.

[3]According to the same statistics, only 14 (or 3%) of 615 insufficient authorization vulnerabilities were detected by automatic scanning. See also [2].

introducing a use case graph, which represents the web application logic and contains important information about actions that are possible within it. Then we traverse this graph performing actions specified by the graph vertices. After each step of the traverse, the state of the web application changes and modified differential analysis is performed. The selected traversal order guarantees that we check each possible action within the application.

## II. Understanding access control vulnerabilities

There are different types of authorization flaws. To be able to evaluate different techniques for their detection capabilities, we point out the following types, based on the root cause[4]:

### A. Privileges under user control

When a web application makes assumptions about user privileges based on unvalidated input, it can be used by an attacker to escalate his privileges. The sample defect of this type is the ability to set `"editMode=true"` parameter in the request, resulting in the ability to edit any document.

Although this vulnerability may exhibit itself in many requests, any of them provide enough evidence about the defect. Thus, discovering n+1 instead n instances is not that much helpful.

### B. Missing access control list entry

When application-wide access control is implemented using a blacklist approach, the missing blacklist entry usually results in an authorization vulnerability. An illustrative example is an Apache-based webserver that lacks .htaccess protection on some folders, leaving admininstrative functionality exposed to public viewers.

Flaws of this type are actually misconfigurations, which are introduced during the deployment phase. They are also identified by any instance, avoiding the need to enumerate all particular cases[5].

### C. Insufficient access control on certain execution paths

When access control checks are distributed within the web application, some execution paths may not be covered by this checks. In this case, enumerating single instances makes sense, as different instances represent distinct execution paths.

[4]We do not claim to have built an exhaustive taxonomy of authorization flaws; these vulnerability types are informal and their mutual exclusion is in question. However, this grouping is useful to outline the capabilities of access control testing methods.

[5]The task of identifying all misconfiguration cases may be challenging as well, but it is more reasonable to review the configuration, once the black-box analysis discovers the root cause

## III. Motivating Example

### A. Sample web application

In order to illustrate the shortcomings of existing "differential analysis" approaches let us have a look at a typical Learning Management System (LMS). Table I describes user actions that are possible within the system ("login" and "logout" actions are omitted for brevity).

Let us suppose that this application contains the following authorization flaws:

1) Student can delete any course that is not assigned to anyone.
2) Manager can assign course to any student, not only to the student assigned to him.
3) The "view course" page is viewable by anyone.

Furthermore, we assume that the system is in the following state:

- There are four registered users, `root` of the role `Admin`, `instructor` of the role `Manager` and $\texttt{student}_{1,2}$ of the role `Student`;
- $\texttt{student}_1$ is assigned to `instructor`;
- there is one course uploaded; this course is assigned to $\texttt{student}_1$.

### B. Existing approach

Existing methods for black-box access control testing are variations of the technique called *differential analysis*. This method suggests that, in order to discover authorization flaws, web application should be browsed on behalf of each user (including the "public" user, which means browsing without credentials) and faced URLs should be recorded (we will refer to this as the *crawling phase*). In our case this will result in five sitemaps: $\mathrm{URL}_{\texttt{root}}$, $\mathrm{URL}_{\texttt{instructor}}$, $\mathrm{URL}_{\texttt{student}_1}$, $\mathrm{URL}_{\texttt{student}_2}$ and $\mathrm{URL}_{\texttt{public}}$.

The next step is to carry out accessibility tests for each pair of users, trying to perform actions that were accessible to one user on behalf of the another. Considering our sample LMS, we should try to access ("$\setminus$" denotes set subtraction):

1) links from $\mathrm{URL}_{\texttt{root}} \setminus \mathrm{URL}_{\texttt{instructor}}$ on behalf of `instructor`;
2) links from $(\mathrm{URL}_{\texttt{root}} \setminus \mathrm{URL}_{\texttt{student}_1}) \cup (\mathrm{URL}_{\texttt{instructor}} \setminus \mathrm{URL}_{\texttt{student}_1})$ on behalf of $\texttt{student}_1$;
3) links from $(\mathrm{URL}_{\texttt{root}} \setminus \mathrm{URL}_{\texttt{public}}) \cup (\mathrm{URL}_{\texttt{instructor}} \setminus \mathrm{URL}_{\texttt{public}}) \cup (\mathrm{URL}_{\texttt{student}_1} \setminus \mathrm{URL}_{\texttt{public}})$ without credentials;
4) links from $\mathrm{URL}_{\texttt{student}_1} \setminus \mathrm{URL}_{\texttt{student}_2}$ on behalf of $\texttt{student}_1$;
5) links from $\mathrm{URL}_{\texttt{student}_2} \setminus \mathrm{URL}_{\texttt{student}_1}$ on behalf of $\texttt{student}_2$.

| User role | Available actions |
|---|---|
| Admin | Create manager, create student, delete student, delete user, assign a student to manager, revoke a student from manager. |
| Manager | Upload a course, assign a course to student, revoke a course from student, delete course; |
| Student | browse a course. |

TABLE I
AVAILABLE ACTIONS FOR A SAMPLE LMS WEB APPLICATION

During steps 1–3, vertical privilege escalation will be detected, whereas steps 4 and 5 test for horizontal escalation.

### C. Discussion of the existing approach

While the described method does very well in simple web applications, in complex systems with lots of interdependent actions it typically fails to detect a large amount of vulnerabilities for two major reasons.

*1) Incomplete web interface coverage:* This problem arises because the described method does not specify the exact order in which these URL collections should be gathered. In our case, if we collect URLs for `root` first, we may occasionaly trigger the "revoke student from manager" action, which will prevent finding the "assign course to student" link in `instructor`'s web interface. Also, the web application state[6] in which we should start gathering the URL collections is not specified. In our example, while enumerating resources and actions visible to `instructor`, the crawler may overlook the "assign course to student" action, because $\texttt{student}_1$ is already assigned the only course that is uploaded at the moment.

*2) Incorrect testing conditions:* Even if the URL is successfully collected, failure to get unauthorized access to it does not necessarily mean it is not flawed. Suppose that we encountered the "assign course to student" link, but later, during browsing, triggered the "delete course" action. When we later try to perform the course assignation on behalf of the student, it may fail not because the access is restricted, but simply because it is not possible to assign a course which is deleted.

Thus, applying differential analysis to our sample LMS will likely fail to detect vulnerabilities 1 and 2, identifying only the publicly-visible "view course" page.

In contrast to well-known difficulties which are encountered during automated or semi-automated black-box web application scanning, (form filling, triggering javascript events in needed order etc.), these are not technical restrictions, but rather limitations of the approach. While it is possible to get over the first limitation by making use of human-directed crawling, manually checking preconditions of each action during the testing phase dramatically increases complexity of the access control audit.

Thus, although typically well suited to discover vulnerabilities of types "A" and, in simplest cases, "B", this method is almost unusable for thorough access control testing in complex systems[7].

## IV. PROPOSED APPROACH

The basic idea of our approach is to perform state-preserving differential analysis in several web application states.

As iterating through all possible states is not an option, we need to carefully select the states in which to perform testing, providing fair test coverage and keeping a reasonable degree of complexity at the same time.

To do this, we introduce the use case graph, which is traversed in a special way. We then execute actions as specified by the traverse, performing URL gathering and accessibility testing after each state change. State-changing actions are not performed during the URL gathering step so that the web application state remains unchanged. This ensures correct testing conditions during accessibility testing.

### A. Selecting appropriate states and ensuring completeness

To get sensible results, we do not have the option of testing the possibly infinite number of valid HTTP requests. Instead, we divide all possible HTTP requests into classes, which aggregate requests with the same semantics. These classes will be called **actions**. For example, all HTTP requests that perform site search will belong to the "search" action whereas all requests that result in forum message creation will form the "create message" class.

Furthermore, we will utilize user roles instead of separate users. Indeed, the number of web application users is arbitrary, new users may be created on the fly and different users of the same role typically have similar privileges so testing all of them is meaningless.

[6]Web application *state* is determined by its runtime environment, including database and filesystem contents, RAM etc.

[7]This is confirmed by the aforementioned WASC statistics [1]: only 3% of applications containing access control flaws were identified during automated scanning

| # | Use Case Name | Depends On | Cancels |
|---|---|---|---|
| **Public** | | | |
| 1 | Browse public resources | - | - |
| **Admin** | | | |
| 2 | Login | 1 | - |
| 3 | Create Manager | 2 | - |
| 4 | Create Student | 2 | - |
| 5 | Assign Student to Manager | 3, 4 | - |
| 6 | Revoke Student from Manager | 5 | 12, 13 |
| 7 | Delete Manager | 3 | 7, 10 - 15 |
| 8 | Delete Student | 4 | 8, 12, 13, 16 - 18 |
| 9 | Logout | 2 | 9, 3 - 8 |
| **Manager** | | | |
| 10 | Login | 3 | - |
| 11 | Upload Course | 10 | - |
| 12 | Assign Course to Student | 11, 5 | - |
| 13 | Revoke Course from Student | 12 | 12, 17 |
| 14 | Delete Course | 11, 13 | 12, 14 |
| 15 | Logout | 10 | 11 - 15 |
| **Student** | | | |
| 16 | Login | 4 | - |
| 17 | Browse Course | 12, 16 | - |
| 18 | Logout | 16 | 17 |

TABLE II
SAMPLE LMS: USE CASE DEPENDENCIES AND CANCELLATIONS

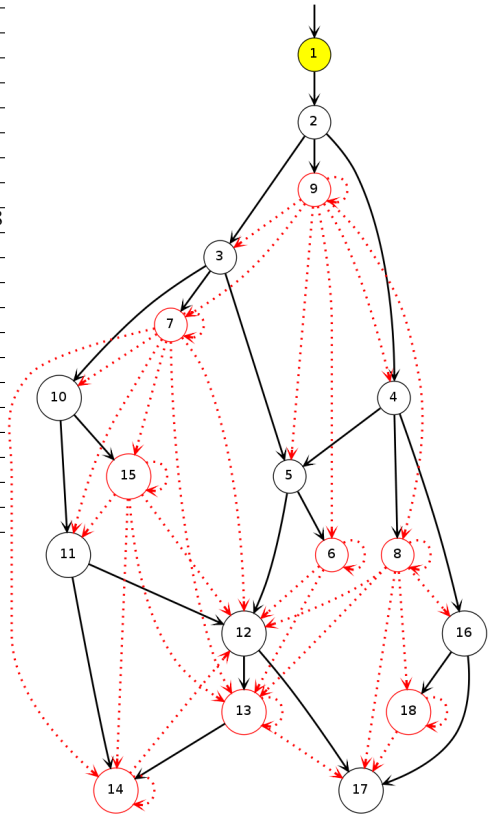


Fig. 1. Use case graph for sample application. Solid edges represent dependencies, dotted edges represent cancellations.

The variety of user roles, on the contrary, is typically determined during the web application design. We also introduce partial order on the set of roles in order to avoid testing the access to actions of lower-privileged users on behalf of the user with higher privileges.

The concepts of action and role may be combined into the notion of **use case**, which stands for *an action performed by a user of a given role*.

We can now switch from trying to cover all possible HTTP requests, which is unfeasible, to testing, at least once, all use cases of the web application. Given that access control is performed based on the use cases, as opposed to treating each request in its own way, we will discover all possible authorization flaws in this manner.

### B. Use case dependencies and cancellations

In a typical web application with nontrivial logic, actions within the application are interdependent (e.g. we should post a message prior to deleting it). We therefore introduce use case **dependencies**. Use case $A$ is considered dependent on use case $B$ when we must perform $B$ in order to be able to perform $A$.

We also introduce use case **cancellations**. Use case $A$ is said to cancel use case $B$, when, after performing $A$ in the state where it is possible to perform $B$, we may lose this opportunity.

Table II enumerates use case dependencies and cancellations for the sample web application introduced above.

We are now able to build a use case graph, including dependencies and cancellations. The graph for the sample application is shown in Figure 1.

### C. Traversing the use case graph

To build the desired sequence of states, we traverse the use case graph in the following way:

1) at the beginning of the traversal, use cases with no prerequisites are available for execution;
2) once a use case is executed, more use cases may become available if their requirements are satisfied;
3) when selecting which use case to execute, we primarily select use cases that do not cancel other use cases. Among them, we pick such a use case that satisfies the maximum number of unsatisfied dependencies.
4) we keep executing use cases until all of them are executed at least once.

### D. Complete testing algorithm

We propose the following procedure for testing the application's access control:

1: Enumerate use cases and roles in the application.
2: Specify use case dependencies and cancellations and build the use case graph.
3: Create two groups of users, each containing a user for every role.
4: Traverse the graph as specified in IV-C to get the use case execution list $UCL$.
5: **for all** $u = (action_i, role_i) \in UCL$ **do**
6: let $user_{1,i}$ be the user of the role $role_i$ from the first group;
7: execute $action_i$ on behalf of $user_{1,i}$;
8: perform crawling, enumerating accessible resources and operations for each user. While doing this, only execution of state-preserving actions is allowed.
9: **for all** $role_1, role_2$ **do**
10: **if** $role_1$ is not less privileged than $role_2$ **then**
11: let $user_1$ be the user of the role $role_1$ from the **first** group;
12: let $user_2$ be the user of the role $role_2$ from the **second** group;
13: let $sitemap_1, sitemap_2$ be the URL collections for $user_1$ and $user_2$, gathered during crawling;
14: try to perform actions from $sitemap_1 \setminus sitemap_2$ on behalf of $user_2$;
15: record any successfully performed action as a authorization flaw;
16: **end if**
17: **end for**
18: **end for**

## V. Implementation details and discussion

We implemented the proposed approach in a proof-of-concept tool called AcCoRuTe (Access Control Rule Tester). The workflow is divided into two stages: information gathering and scanning.

### *Information gathering*

During the first step, a self-developed Firefox extension assists the operator in building the use case graph. Actions are recorded as the operator performs them using the browser. The operator may then use them to build the use case graph. He also provides additional information that will be passed to the scanner (e.g. list of web application roles and users, the crawling scope, etc).

The process of use case graph creation cannot be performed in a fully automated way for a black-box scenario. We can nevertheless significantly reduce the complexity of this task. First of all, most of the state-preserving actions may be excluded from the graph. Indeed, these actions cannot be dependencies or cancel other actions. Furthermore, if the state-preserving action has only one dependency, we can safely assume that this dependency will be met at least once during the use case graph traversal.

We should also note that providing an incomplete use case graph may reduce test coverage (resulting in decreased numbers of discovered and tested actions), but will not influence the correctness of the analysis (i.e. the discovered actions will still be tested in proper states). Thus, when creating the use case graph of a very large web application, the operator may choose to enumerate use cases at the coarse-grained level (e.g. only enumerate dependencies and cancellations for the most important actions), possibly reducing the coverage. However, in order to correctly distinguish between state-changing and state-preserving actions during the scan, the operator should record a representative for each state-changing action.

### *Scanning*

Gathered information is passed to the second part of the tool, the scanner. The scanner is a standalone Java application based on HtmlUnit[7].

According to the presented approach (see section IV), the scanner navigates the web application as follows:

1) recursively crawls the web application, building "site maps" for each user (only state-preserving actions are allowed);
2) performs accessibility tests;
3) changes web application state by issuing one of the state-changing requests according to the use case graph or returns, if the traversal is complete;
4) proceeds with step 1 in the changed state.

*Combining requests into actions:* During the second and third steps the scanner needs a way to detect similar HTTP requests and combine them into actions. If the sitemaps of two different users contain requests that differ in detail, but share meaning (e.g. two forum posts with different text), they should be considered equivalent.

In order to test requests for similarity, we represent each of them as a list of parameters. Every parameter is then assigned a meaning which is inferred from the web application interface: for example, user-modifiable form fields map into user-controllable parameters, whereas hidden fields are considered "automatic" (i.e. out of user control). The current implementation also allows the operator to manually map parameter name/location pairs to their meanings in order to identify session/CSRF tokens and volatile

parameters (e.g. document identifiers)[8]. When testing for similarity, the scanner ignores user-controllable parameters and session identifiers because they do not contribute to the meaning of the request.

*Preventing uncontrolled state changes:* As we already mentioned, only state-preserving requests should be issued during the crawling step. To recognize state-changing requests the scanner tests them for similarity with actions recorded by the operator.

During the testing step the state of the web application may also change, which means that a state-changing action is vulnerable. In this case, the scanning process cannot be continued, because the state of the application after successful exploitation is unpredictable. Therefore, if the scanner detects vulnerable state-changing action, it stops execution. This action is then added to scanner "suppression" list (i.e. list of actions that will not be tested), the application is reset to its initial state by the operator and the scan is launched again.

*Determining successfully performed actions:* During accessibility testing, we need a way to understand whether a request was successful or not. The usual approach here is to perform pattern matching in order to identify "access denied"-like HTTP responses based on operator input, which is prone to both false positives and false negatives. Instead, we compare received response with the valid response to the similar request that was collected during previous steps and use pattern matching as a fallback, if this information is not available. To test responses for similarity we compute normalized edit distance [8] and introduce a configurable similarity threshold. Although this approach does well in typical scenarios, it may fail to detect performed actions in some cases (e.g. when the web application responds as if the access was denied, while nevertheless performing the action).

## VI. Evaluation

We evaluated our tool and our approach on a real-world web application (*Easy JSP Forum*) obtained from SourceForge repository, which is a typical message board written in JSP. This application was previously analyzed by Felmetsger et al [9], who discovered two access control vulnerabilities, namely that any authenticated user is able to delete and modify any post in the forum.

Using our tool we were able to find one known vulnerability[9] and three previously unknown vulnerabilities in this application: **any moderator can delete or modify any forum as well as assign it to any user**.

The application contains a management interface, which can be accessed by forum administrators and moderators. The interface allows assigning a forum to a moderator (corresponding form is shown only to administrative users) and contains links to the "forum details" page where a forum can be edited or deleted. Moderators are only allowed to edit or delete forums that belong to them, and for them only allowed "edit forum" links are shown. However, when a delete/modify request is issued, the forum application fails to check whether the moderator owns the forum he is about to modify or delete. The individual "edit forum" pages also lack this check.

Furthermore, the "assign forum to moderator" action, which must be available only to forum administrators is in fact accessible by moderators as well. The corresponding execution path fails to ensure that the session belongs to the administrator (although it checks whether the user has at least moderator privileges, therefore regular users cannot exploit this vulnerability)

Our tool reported this as four vulnerable actions: three state-changing actions (forum deletion, modification and assignment) and one state-preserving (the "edit forum" page itself).

Our tool also incorrectly reported the "view profile" action as vulnerable, because the "view profile" link for a user is shown to other users only if that user posted at least one message to the forum, whereas it is always shown to the user himself. This is the only false positive reported.

Also, AcCoRuTe failed to detect one of the previously-known vulnerabilities which allows post modification. That was due to limitations of its crawling engine (javascript-generated popups are not always correctly processed), and it will be fixed soon.

Some details about performed scan iterations are presented in Table III. All tests were performed on a PC with an Intel Xeon CPU (2.33 GHz) with 16 GB of RAM. The *Pages Fetched* and *Pages Parsed* columns represent the number of fetched URLs (including images, scripts and style sheets) and the number of actually parsed pages, respectively. The *Forms Filled* column denotes the number of forms which were automatically filled in during the crawl (note that each form was filled several times by different users in different states). The *Checks Performed* column shows the number of the accessibility tests carried out, the *Raw Alerts* column shows the number of successful tests, whereas *Unique Alerts* shows the number of distinct vulnerable actions as reported by the tool.

The amount of human work required to build the

[8]It is possible to automate this process (e.g. by repeatedly fetching the index page of the application to discover session token name and location).

[9]The tool discovered the vulnerable "delete post" action but failed to discover the "edit post" vulnerability.

| Iteration[a] | Pages Fetched | Pages Parsed | Forms Filled | Checks Performed | Raw Alerts | Unique Alerts[b] | Vulnera-bilities | False Positives | Runtime (min:sec) |
|---|---|---|---|---|---|---|---|---|---|
| **First** | 499 | 127 | 175 | 302 | 22 | 5 | 4 | 1 | 3:50 |
| **Second** | 944 | 254 | 315 | 512 | 25 | 1 (2) | 1 (1) | 0 (1) | 7:50 |
| **Third** | 2566 | 709 | 833 | 1323 | 61 | 0 (2) | 0 (1) | 0 (1) | 15:52 |
| **Total** | | | | | | 6 | 5 | 1 | 26:52 |

TABLE III
EXPERIMENTS ON *Easy JSP Forum*

[a]Three scanner runs were required to completely test this web application: after discovering a state-changing vulnerable action the scanner stops, discovered vulnerabilities are added to the suppression list, web application is reset to its initial state and the scanner is launched again (possible ways to recover after discovering state-changing vulnerabilities is discussed in section V).

[b]For the second and third runs of the scanner the number of alerts that duplicate alerts reported in previous runs is shown in brackets

use case graph cannot be measured objectively. During our experiments it took about 30 minutes to record the needed use cases and specify their dependencies and cancellations for an operator familiar with forum functionality. The recorded use case graph for *Easy JSP Forum* contained 11 nodes and 22 edges (8 dependencies and 14 cancellations).

## VII. LIMITATIONS

Both, presented approach and its implementation have certain limitations which are briefly enumerated in the current section.

### A. Limitations of the approach

*1) Vulnerability types:* The suggested approach ensures that every use case will be discovered and tested at some stage of graph traversal. This means that we will try every use case at least once. As mentioned above, this is enough to find all authorization defects only if we assume that the access control flaws that we are dealing with are of a certain type. More specifically, we assume that, if

- unauthorized access to a given use case $uc = (action, role_{victim})$ by a user of a given role $role_{attacker}$ is possible
- AND the web application in its current state allows one of the users of $role_{victim}$ to perform the $action$,

we will be able to reproduce unauthorized access in the current state by executing $action$ on behalf of any user of $role_{attacker}$.

If, for example, the vulnerability appears only when deleting the 20th blog post, or only if three or more customers are concurrently shopping, the presented approach will probably fail to detect them.

*2) Hidden functionality:* Another limitation arises from the test case selection: the proposed approach will not discover actions and data that are not explicitly linked from the user's web interface (e.g. database backups stored online). There are various solutions for this problem, including fuzzing (stateful or stateless), which can be combined with our approach to improve its completeness. This is, however, beyond scope of this paper.

### B. Limitations of the implementation

The current implementation has several technical limitations which are typical for automated black-box crawlers. Although the tool provides reasonable javascript support, it may still fail to detect some links, especially when non-trivial user interactions are needed to trigger respective requests (e.g. multiple mouse movements and clicks).

The accessibility test may also fail to detect that the tested action was actually carried out (e.g. when the "access denied" message was shown, but the action was nevertheless performed).

The implementation currently lacks AJAX support, which is left for future work.

The amount of manual operator work is still high especially in case of very large web applications. Some possible ways to further automate the approach were already discussed.

Finally, the efficiency of the scanner should be improved in order to perform analysis of large-scale web applications.

## VIII. RELATED WORK

First, our work is related to the research on inferring program specifications in order to find vulnerabilities. One of the first techniques that use inferred specifications in order to find application-specific errors was developed in the work by Engler et al. [10]. Using a number of pre-defined templates, the authors infer specifications in the form of programmer "beliefs". The code is then checked for contradictions with these beliefs and statistical analysis is used to distinguish between valid beliefs and coincidences. Another work by Tan et al. [11] focused on detecting missing security checks using specifications automatically extracted from the source code. Kremenek et al. [12] also leveraged static source code analysis to extract program specifications and find bugs.

Various authors also used dynamic analysis to derive program specifications, which are then either used for intrusion detection ([13], [14]), or statically checked (as in the research by Nimmer and Ernst [15].

The Waler tool (Felmetsger et al., [9]) also explores the combination of dynamic and static analysis in order to detect vulnerabilities. They infer specifications by dynamically analyzing the web application's "normal operation", and then they use static analysis to find execution paths that violate the specifications. This approach is similar to ours in the sense that both works try to detect web application vulnerabilities by checking the inferred specifications. Our work, however regards the web application as a black box, inferring specifications from its interface and uses accessibility tests to check the inferred specifications. Our approach requires a significant amount of operator work in order to express the web application logic as a use case graph, whereas the combination of dynamic and static analysis probably offers a higher level of automation. However, the completeness of the analysis performed by Waler seems to depend significantly on how the normal behavior is covered during the invariant extraction. This is illustrated by *Easy JSP Forum* web application, which contains at least three vulnerabilities that were not reported by Waler, possibly because of an incomplete set of inferred invariants (see Section VI).

Our research is also related to several works that use black-box analysis to detect authorization flaws in web applications. The proposed approaches include forceful browsing (i.e. manual testing), differential analysis and fuzzing.

Segal [6] describes the aforementioned differential analysis technique. The shortcomings of this method, which prevent it from discovering complex access control flaws in large applications, are described in Section III. Book [16] by Stuttard and Pinto contains a complete overview of black-box web application security analysis. In respect to attacking authorization, the book suggests a variation of differential analysis, which is afflicted by the above mentioned limitations. Book [5] also overviews black-box analysis techniques and offers differential analysis as an alternative to manual access control verification.

The task of discovering hidden (not linked explicitly from the web interface) data and functionality of the web application is also related to our work because access to hidden content is typically not restricted ("security by obscurity"). This problem is typically approached by fuzzing (i.e. educated and optimized guesswork). For example, OWASP DirBuster tool [17] uses dictionaries to guess locations of hidden files and directories. Discovering hidden content is out of scope of our approach.

Our work is also related to execution-after-redirect (EAR), a web application vulnerability researched by Doupe and Boe [18], that may lead to broken access controls. In case of unauthorized access, the vulnerable application generates redirect but fails to stop further request processing. Assuming that the request processing generates output, which is sent back in the body of the redirect, and assuming that this output is similar to that of the authorized user, our approach remains applicable for this case.

## IX. Conclusion

In this paper, we introduced a method to detect access control flaws in web applications by black-box analysis. We extend the widely-used "differential analysis" to take web application state into account. In our approach, we construct the web application use case graph with assistance of a human operator, traverse it in a specific order to get the sequence of use cases, iterate through this sequence and apply differential analysis at each step of the traverse.

We implemented the proposed approach in a tool that we named AcCoRuTe, and it was able to identify previously unknown access control vulnerabilities in a real-world web application.

## References


[1] W. A. S. Consortium, "Web application security statistics 2008," http://projects.webappsec.org/f/WASS-SS-2008.pdf, Web Application Security Consortium.

[2] J. Grossman, "WhiteHat Website Security Statistics Report," http://www.whitehatsec.com/home/assets/WPStatsreport_100107.pdf, 2007.

[3] A. Masood, "Scalable and effective test generation for access control systems," 2006.

[4] 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*. ACM, 2006, pp. 752–753.

[5] J. Scambray and M. Shema, *Hacking exposed: Web applications*. McGraw-Hill Osborne Media, 2002.

[6] O. Segal, "Automated testing of privilege escalation in web applications," *Watchfire*, 2006.

[7] "HtmlUnit." [Online]. Available: http://htmlunit.sourceforge.net

[8] A. Weigel and F. Fein, "Normalizing the weighted edit distance," in *Pattern Recognition, 1994. Vol. 2-Conference B: Computer Vision & Image Processing., Proceedings of the 12th IAPR International. Conference on*, vol. 2. IEEE, 1994, pp. 399–402.

[9] V. Felmetsger, L. Cavedon, C. Kruegel, and G. Vigna, "Toward automated detection of logic vulnerabilities in web applications," in *USENIX Security*, 2010.

[10] D. Engler, D. Chen, S. Hallem, A. Chou, and B. Chelf, *Bugs as deviant behavior: A general approach to inferring errors in systems code*. ACM, 2001, vol. 35, no. 5.

[11] L. Tan, X. Zhang, X. Ma, W. Xiong, and Y. Zhou, "Autoises: Automatically inferring security specifications and detecting violations."

[12] T. Kremenek, P. Twohey, G. Back, A. Ng, and D. Engler, "From uncertainty to belief: Inferring the specification within," in *Proceedings of the 7th symposium on Operating systems design and implementation*. USENIX Association, 2006, pp. 161–176.

[13] M. Bond, V. Srivastava, K. McKinley, and V. Shmatikov, "Efficient, context-sensitive detection of semantic attacks," Citeseer, Tech. Rep., 2009.

[14] A. Baliga, V. Ganapathy, and L. Iftode, "Automatic inference and enforcement of kernel data structure invariants," in *2008 Annual Computer Security Applications Conference*. IEEE, 2008, pp. 77–86.

[15] J. Nimmer and M. Ernst, "Static verification of dynamically detected program invariants:: Integrating daikon and esc/java," *Electronic Notes in Theoretical Computer Science*, vol. 55, no. 2, pp. 255–276, 2001.

[16] D. Stuttard and M. Pinto, *The Web Application Hacker's Handbook: Discovering and Exploiting Security Flaws*. Wiley Publishsing, 2007.

[17] "OWASP DirBuster." [Online]. Available: https://www.owasp.org/index.php/Category:OWASP_DirBuster_Project

[18] A. Doupe, "Overview of execution after redirect web application vulnerabilities," http://adamdoupe.com/overview-of-execution-after-redirect-web-appl.