SN Computer Science (2022) 3:376
https://doi.org/10.1007/s42979-022-01271-1

ORIGINAL RESEARCH

Automating the Detection of Access Control Vulnerabilities in Web Applications

Marc Rennhard[1] · Malte Kushnir[1] · Olivier Favre[1] · Damiano Esposito[2] · Valentin Zahnd[2]

Received: 24 September 2021 / Accepted: 24 June 2022

Abstract

The importance of automated and reproducible security testing of web applications is growing, driven by increasing security requirements, short software development cycles, and constraints with respect to time and budget. Existing automated security testing tools are already well suited to detect some types of vulnerabilities, e.g., SQL injection or cross-site scripting vulnerabilities. However, other vulnerability types are much harder to uncover in an automated way. One important representative of this type are access control vulnerabilities, which are highly relevant in practice as they can grant unauthorized users access to security-critical data or functions in web applications. In this paper, a practical solution to automatically detect HTTP GET request-based access control vulnerabilities in web applications is presented. The solution is based on previously proposed ideas, which are extended with novel approaches to enable completely automated access control testing with minimal configuration effort, which in turn enables frequent and reproducible testing. An evaluation with seven web applications based on different technologies demonstrates the general applicability of the solution and that it can automatically uncover most access control vulnerabilities while keeping the number of false positives low.

Keywords Automated testing · Web application security testing · Access control testing · Black box security testing · Dynamic web application security testing

Introduction

This article is an extended and revised version of our work published in [1]. Compared to the original paper, our work has been improved in three main areas. First, the overall solution approach has been adapted (see "Solution approach"), which results in an improved vulnerability detection performance. Second, the evaluation has been extended (see "Evaluation") from the original four applications and now includes seven applications. And third, the evaluation results are discussed in much more detail, which results in an improved understanding of both the effectiveness and the limitations of the presented approach.

Web applications are prevalent in today's world and are used for a wide range of services. This includes typical end-user services such as information portals, e-banking, e-shopping and social networks, but web applications are also used for many other purposes such as configuring devices attached to the Internet (e.g., switches, routers and Internet of Things (IoT) devices) or controlling and monitoring highly complex industrial processes as part of SCADA systems.

This article is part of the topical collection "Information Systems Security and Privacy" guest edited by Steven Furnell and Paolo Mori.

✉ Marc Rennhard
rema@zhaw.ch

Malte Kushnir
kusn@zhaw.ch

Olivier Favre
favr@zhaw.ch

Damiano Esposito
Damiano.Esposito@scanmeter.io

Valentin Zahnd
Valentin.Zahnd@scanmeter.io

[1] School of Engineering, Zurich University of Applied Sciences, Technikumstrasse 9, Winterthur 8401, Switzerland

[2] Scanmeter GmbH, Bellariastrasse 12, Zurich 8002, Switzerland

Published online: 15 July 2022

Many web applications provide access to security-critical (sometimes also safety-critical) data or functions and correspondingly, they are often attractive targets and are frequently attacked [2]. Therefore, it is of highest importance to implement and configure them in a secure way to increase their resistance to attacks. Ideally, this includes employing a secure development life cycle, where security measures are appropriately considered during all phases of the development process. One important part of this is security testing. Security testing can be performed in different ways, ranging from completely manual methods (e.g., manual source code analysis), to semi-automated methods (e.g., analyzing a web application using an interceptor proxy), to completely automated ways (e.g., analyzing a web application using a vulnerability scanner). Ideally, at least parts of security testing should be automated as this increases efficiency and enables continuous and reproducible security tests, which is getting more and more important in light of today's short software development cycles.

To automate security testing of web applications, different methods can be used. The most popular approaches include static and dynamic code analysis and vulnerability scanning. Web application vulnerability scanners test a running application "from the outside" by sending specifically crafted requests to the target and by analyzing the received response. Various vulnerability scanners are available as either commercial products or open source projects (see [3] for an overview) and while these tools have different strengths and weaknesses [4], they are in general well suited to detect some types of web application vulnerabilities.

Web application vulnerability scanners are especially well suited to detect *technical vulnerabilities*, which include, e.g., SQL injection (SQLi), cross-site scripting (XSS) or cross-site request forgery (CSRF) vulnerabilities. The reason for this is that in the case of technical vulnerabilities, the responses received from the web application during vulnerability scanning often provide good evidence to determine whether a particular vulnerability exists or not. For instance, if the vulnerability scanner sends JavaScript code in a parameter of a GET or POST request and if the returned HTML response includes this JavaScript code in its original (non-sanitized) form, then it is basically proven that the web application is prone to a reflected XSS vulnerability.

Logical vulnerabilities, on the other hand, are much more difficult to detect for vulnerability scanners. Among the most prominent of these logical vulnerabilities are access control vulnerabilities, which allow users of a web application to access data or functions that should not be accessible for them. At first glance, it seems to be relatively easy for a vulnerability scanner to detect such vulnerabilities as it is usually possible to configure the required credentials (e.g., usernames and passwords) of different users so that the scanner can interact with the web application under test using the provided identities. However, the vulnerability scanner will not be able to assess whether the access control rules are enforced correctly unless it has additional information about how the web application should work correctly. To illustrate this, assume an e-shop should provide the functionality to create and modify products only to authenticated users with the role seller. If the scanner now analyzes the e-shop using the credentials of a user that has the role customer and if the scanner manages to modify a product, then this is an obvious access control vulnerability and could easily be detected by a human. But for the scanner, it is virtually impossible to make a correct decision because it does not understand who is allowed to do what in the application under test.

At the same time, access control vulnerabilities are highly relevant and critical in practice. On the newest OWASP list of the ten most critical vulnerability types in web applications (OWASP Top 10 2021 [5]), this vulnerability type has moved up to the first position. Consequently, there is a strong desire for solutions that can at least partly automate the detection of access control vulnerabilities. Several approaches have been proposed in the past (see "Related work"), including tools to assist a manual security tester to do access control tests more efficiently, methods to extract access control models from the source code so it can be verified, methods where the access control model has to be supplied to the scanner so it can verify its correct implementation, and fully automated solutions where a scanner first tries to determine the access control model and then tries to verify it by interacting with the web application.

While all these proposals have their merits, they usually have limitations that prevent their wide applicability in practice. For instance, approaches that still require lots of manual work are simply not suited for frequent and reproducible testing of web applications. Also, requiring a formal specification of an access control model that can be supplied to the scanner is rarely an option as such models are usually not available in practice. Furthermore, some approaches are dependent on the web application technology (i.e., programming language or web framework) that is used, which usually means they have to be adapted if different technologies are used. In addition, access to the source code is not always granted when a web application of a third party is analyzed. And finally, most of the proposed approaches were evaluated using a very limited set of web applications (often just one), which means that there are several open questions with respect to the general applicability and usefulness of the approaches.

In this paper, we present a solution that overcomes the limitations of previously proposed approaches. Our solution is based on existing ideas, which are extended and optimized to enable completely automated, black box-based detection of access control vulnerabilities in the context of HTTP GET requests. The solution is applicable with minimal

configuration effort to a wide range of web applications without requiring access to the source code and without requiring the availability of a formal access control model. Our evaluation demonstrates that the solution can indeed be applied to various types of web applications and that it can uncover several vulnerabilities while keeping the number of false positives low. As a result of this, the solution can effectively be used in various scenarios, e.g., by penetration testers to quickly and efficiently detect access control vulnerabilities in a web application or by companies or third-party security testing service providers to test web applications for access control vulnerabilities continuously and in a reproducible way, both during development and operation.

The remainder of this paper is organized as follows: The next section provides an introduction to access control vulnerabilities in web applications. The subsequent sections describe our solution to detect such vulnerabilities in an automated way and the evaluation results. Then the solution approach and the evaluation results are discussed and directions for future work are given. The penultimate section covers related work and the last section concludes this work.

Access Control Vulnerabilities

Access control vulnerabilities in web applications can be divided into two main categories: function-level and object-level vulnerabilities. Function-level access control vulnerabilities occur if a web application does not sufficiently check whether the current user is authorized to access a specific function (which often corresponds to a resource in the web application). For instance, assume that the URL https://www.site.com/admin/viewcustomers is intended to be accessible only by administrators of an e-shop to view the registered customers. In this case, the resource admin/viewcustomers identifies the function to view customers and if a non-administrative user of the web application manages to successfully access this function, then this corresponds to a function-level access control vulnerability. Object-level access control vulnerabilities occur if a user gets unauthorized access to objects within a web application. For instance, assume that a seller in an e-shop has legitimate access to edit his own products that the product with *id=123* belongs to him, and that the URL https://www.site.com/editproduct &id=123 grants him access to edit the product. If this seller now changes the *id* in this URL so that it corresponds to a product that belongs to another seller (e.g., using *id=257* so that the resulting URL is https://www.site.com/editproduct &id=257) and if access to edit the product is granted by the web application, then this corresponds to an object-level access control vulnerability.

There are many possible root causes for such vulnerabilities, including coding mistakes (e.g., forgetting to include an authorization check or implementing the check incorrectly), configuration errors (e.g., Cross-Origin Resource Sharing (CORS) misconfigurations in the case of REST APIs), and vulnerabilities in web application server software and web application frameworks [6]. The impact of an access control vulnerability depends on the vulnerable component and can range from information disclosure (e.g., an e-banking customer who can view the account details of other customers) to unauthorized manipulation of data (e.g., a user of a web-based auctioning platform who can modify the bids of other users) up to getting complete access to and control of the affected web application (e.g., an attacker that manages to get access to the function to create a new administrator account in the web application).

While access control vulnerabilities are relatively easy to exploit (e.g., by incrementing an object identifier), they are often difficult to discover. There are some tools that can assist manual penetration testers when trying to find such vulnerabilities (see "Related work"), but these tools have limitations, in particular in the context of large web applications. Large web applications can contain hundreds of functions protected by different kinds of access control mechanisms and typically use multiple user roles and complex permission management systems. The combination of all functions and roles can quickly result in a combinatorial explosion which makes comprehensive manual testing impossible. Many penetration testers, therefore, usually rely on their experience to find access control vulnerabilities rather than systematically testing the entire web application. The obvious problem with this approach is that it usually results in testing only parts of the web application, which means that testing coverage is often low and the probability that possibly serious vulnerabilities are missed is high. This underlines the need for better mechanisms to automate the detection of access control vulnerabilities as presented in this paper, since only a high degree of automation will make it possible to test web applications thoroughly and repeatedly.

To further stress the significance of access control vulnerability testing, it is important to mention that unlike technical vulnerabilities that can often be detected by analyzing requests and looking for typical attack patterns, access control vulnerabilities cannot be detected by looking for anomalies in a request. For instance, if a request contains a parameter id=' UNION SELECT Username,Password from User--, then this can easily be identified as an SQL injection attack due to the very typical attack pattern and correspondingly, a filtering device such as a Web Application Firewall (WAF) can be configured to block such requests. However, with requests that try to exploit access control vulnerabilities, there are no suspicious patterns visible. For instance, if

a seller is allowed to access one of his own products using a parameter *id=278* but then tries to access a product of another seller using *id=315*, then both request look equally innocent and as a result, there's no practical way to define a general rule in a filtering device such as a WAF to identify the second request as malicious.

Solution Approach

The solution presented in this paper focuses on automated detection of access control vulnerabilities using a black box approach. Black box means that the solution analyses a running web application by interacting with it "from the outside" and that it neither requires nor makes use of any internal information such as the source code or access control configurations.

The solution is able to detect function-level and object-level access control vulnerabilities in the context of HTTP GET requests. The main reason for the current focus on GET requests is to minimize the risk of side effects and changes in the application state that usually happen with POST, PUT, PATCH or DELETE requests. These request types typically trigger changes in the data and/or state of an application and, therefore, can alter its appearance and functionality. As a result of this, the application state would constantly change during testing, making it very difficult to compare application behavior when using different users.

The fundamental idea of how our solution determines legitimate accesses in the web application under test is based on the assumption that in most web applications, the web pages presented to a user contain only links, buttons and other navigation elements that can legitimately be used by this user. For instance, navigation elements to access administrative functions in a web application typically only show up after an administrator has successfully logged in and are usually not presented to users that do not have administrator rights. Based on this assumption, web crawling can be used to determine the content which can be legitimately accessed by different users of a web application, as web crawlers mainly follow navigation elements that are presented to a user by the web application.

This basic approach is not novel and there exist publications on it or that use of parts of it (see "Related work"). However, what separates our solution from previous work is that it is designed to be a truly practical approach that is easily applicable to a wide range of web applications, that it is highly automated without requiring manual effort beyond providing a simple configuration file, and that it aims at optimizing automated detection of vulnerabilities with the goal to maximize true positives while minimizing false positives. This is achieved using a strict black box approach and by integrating several novel approaches in our overall solution, in particular by combining crawlers with different strengths (see "Crawling"), using a sequence of several filtering steps to remove HTTP request/response pairs that are not relevant when doing access control tests (see "Filtering"), and using a multi-step validation approach to determine whether a possibly detected access control vulnerability is indeed a vulnerability or not (see "Validating").

Prerequisites

The solution is based on two main prerequisites. The first prerequisite is that one has to know the authentication mechanism used by the web application under test. This can easily be determined by observing the HTTP traffic while accessing the application with a web browser and with the help of an interceptor proxy or developer tools that are available for several web browsers.

Most web applications use either cookie-based authentication with session IDs or token-based authentication with JSON Web Tokens (JWT). Authentication with session IDs is commonly used by applications following a classic architecture and by popular frameworks such as Django, Jakarta EE Spring or Ruby on Rails in their default configuration. Token-based authentication is often used in applications based on a modern architecture where a JavaScript frontend framework such as Angular, React or Vue communicates with a REST API in the backend. The solution presented here supports both these authentication schemes. The second prerequisite is that for each user account which should be included in the analysis, authenticated session IDs or tokens can be created and captured. This can be done using the standard sign-up and login functionality of a web application.

Specifying the authentication mechanism and supplying corresponding authentication information for the users that should be included in the analysis is the minimum configuration required in our solution. Additional configuration is possible and explained later in this section together with a full configuration example.

Overall Workflow

Figure 1 illustrates the overall workflow of the presented solution.

As inputs, the URL of the target web application, the authentication mechanism, and authentication information (e.g., authenticated session IDs, JWTs) of two users, U_1 and U_2, are required. These two users can have different roles (e.g., an administrator and a seller in an e-shop) or they can have the same role (e.g., two sellers in an e-shop that both offer their own products). Note that the workflow described in this and the following subsections is always based on two authenticated users and determines access control

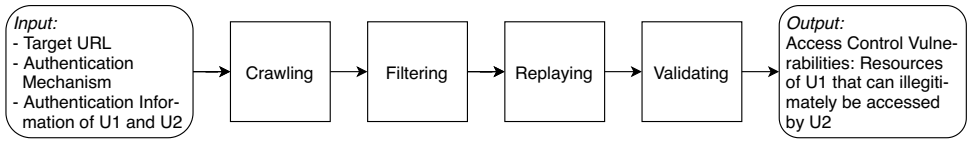

Fig. 1 Access control testing workflow

vulnerabilities where U_2 can illegitimately access resources of U_1. However, this can easily be extended to more than two users and also works if one of the users is an anonymous (unauthenticated) user, as will be explained in "Testing of multiple users and roles".

In a first step, the *crawling component* is used to capture all reachable content of the target web application when using the authentication information of the two users and when using the anonymous user. The data collected by the crawling component is passed to the *filtering component*, which removes data points from the crawling results of U_1 that are not relevant when doing access control tests do determine what resources of U_1 can illegitimately be accessed by U_2. The filtered crawling results are then used as input for the *replaying component*, which replays requests that were captured when crawling with U_1 using the identity of U_2. This delivers the replaying results, which are passed to the *validating component*. This final component analyses the replaying results to determine the actual access control vulnerabilities, i.e., the resources of U_1 that can illegitimately be accessed by U_2. The individual components of this workflow are explained in more detail in the following subsections.

Crawling

The crawling component receives the target URL, the authentication mechanism, and authentication information of two users U_1 and U_2 as inputs and uses two custom crawlers, built with the popular crawling frameworks *Scrapy* [7] and *Puppeteer* [8]. The workflow of this component is shown in Fig. 2.

Combining two crawlers has several benefits. First, coverage improves by merging the results of two crawlers since their underlying detection mechanisms vary. Second, by adding the custom crawler based on Puppeteer, it is possible to capture HTTP requests and responses that are executed as part of JavaScript code due to user interaction with navigation elements in the browser, which are often requests to a REST API. Many applications that are based on a modern architecture dynamically load and modify the Document Object Model (DOM) of a web page and can therefore not be reliably crawled with a crawler based on Scrapy that relies on parsing HTML documents. The custom crawler based on Puppeteer, however, allows to control a real Chrome browser instance and waits for a page to fully load all dynamic content before processing the page for any navigation links and action elements such as buttons.

At first glance, it seems that using the Scrapy-based crawler is redundant to using the Puppeteer-based crawler as everything that is found by the first one should always be also found be the second one. In most cases, this is true. However, the Scrapy-based crawler is much faster than the one based on Puppeteer because the latter always has to wait for several seconds after calling a URL or performing an action to make sure the resulting page has completely been processed by the Chrome browser instance before interacting with it. Therefore, in the case of web applications that follow purely a classic architecture, it is often sufficient to only use the Scrapy-based crawler, which is much faster than using both. With web applications that follow a modern architecture, however, it is always recommended to use both crawlers to maximize the number of resources that are detected during crawling.

Both crawlers are used to crawl the target application three times: One time for each of the users U_1 and U_2 for which authentication information has been provided and once without any authentication information to capture content that is publicly accessible. Since the solution currently focuses on GET requests, no other request types are used during crawling. The output of the crawling component are three lists of HTTP request and response pairs (one for U_1, one for U_2, and one for the anonymous user) that resulted from the three crawling runs. Note that the HTTP responses can include different content depending on the corresponding request, e.g., an HTML document, JSON data received from a REST API, or binary data in case of an image or PDF document. Duplicates created during crawling are removed from the lists, i.e., if a URL is used multiple times when crawling with a certain user, e.g., U_1, the corresponding request and response pair is included only once in the list of U_1.

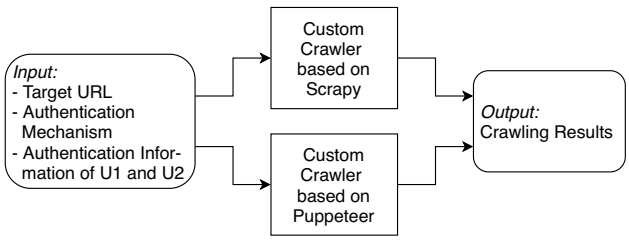

Fig. 2 Crawling component

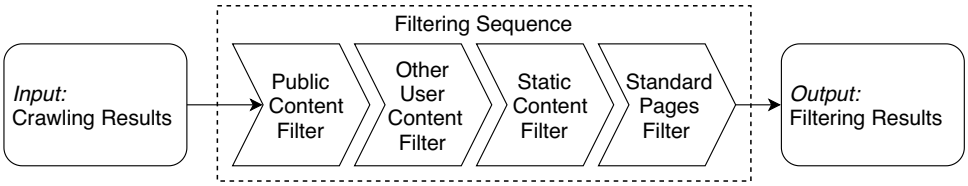

Fig. 3 Filtering component

Filtering

The filtering component uses a sequence of four filters to remove HTTP requests and responses that are not relevant when doing access control tests in the context of U_1 and U_2. These filters analyze all HTTP request and response pairs that were collected during crawling and discard pairs that meet specific criteria. Figure 3 depicts the steps performed during filtering.

The first filter, the *public content filter*, looks for content that is accessible without any authentication. It compares the URLs of the requests in the crawling results of U_1 with the URLs in the crawling results when no authentication information was used. If there is an overlap, the content is considered public information and therefore filtered out from the crawling results of U_1.

The second filter, the *other users content filter*, removes content from the crawling results of U_1 that is also present in the crawling results of U_2. To do so, it compares the URLs of the requests in the crawling results of U_1 and U_2 and if there is an overlap, the content is filtered out from the crawling results of U_1. The rationale here is that if a URL is found during crawling with U_1 and U_2, it is assumed that both users can legitimately access the content and consequently, it must not be considered in the context of access control tests with the current two users and can therefore be filtered out.

Next, the *static content filter* filters out static content from the crawling results of U_1 that is not relevant for access control decisions. This applies mostly to frontend-specific content that is served as static files, e.g., CSS files, JavaScript files, image files, etc. This is a configurable filter that can be supplied with a list of file extensions that mark static content. The filter will then filter out all requests that point to files with these extensions. By default, CSS files, JavaScript files and image files with well-known file endings are filtered.

Finally, the *standard pages filter* filters out specific standard pages from the crawling results of U_1. Many web applications contain pages like "About Us" or contact forms that are typically not relevant in access control testing scenarios. The filter looks for keywords in URLs that indicate such pages. It is preconfigured with a list of common keywords and can optionally be customized. This can be helpful if the web framework that was used to develop the web application is known since many frameworks have well-known paths for standard pages.

In the original paper [1], a fifth filter was used, identified as *content similarity filter*. This filter checked whether the remaining crawling results of U_1 contains HTTP responses that are equal or very similar to each other. This filter mainly aimed at filtering out standard content that is missed by the previous filter. For instance, some applications do not use HTTP status codes to indicate that a page was not found, that a redirect occurred, or that some other error happened, but instead respond with an HTTP 200 status code and a custom error page. This would have been detected by this filter and the corresponding requests and responses would have been removed from the crawling results of U_1. However, further analyses in the context of additional evaluations led us to the conclusion that this filter can possibly do more harm than good. In particular, if multiple URLs in the crawling results point to very similar content that is access control sensitive, such content would be filtered out as well. For instance, assume that when crawling with U_1, two different URLs are found (where the difference can be as small as a query parameter for sorting that is used in one case but not in the other) that both point to the page that shows the full list of registered users in the application. When using the content similarity filter, both URLs would be removed from the crawling results of U_1. However, in case there is an access control vulnerability that allows U_2 to use one of these URLs to illegitimately access the page with all registered user, this could no longer be detected during the subsequent replaying and validating steps as the URL has been removed. To prevent such cases, we decided to no longer use this fifth filter.

The output of the filtering component is a list of filtered HTTP request and response pairs of U_1. Its content is a subset of the original crawling results when using U_1 and it contains the entries that were not removed by any of the four filters of the filtering sequence.

Replaying

The replaying component takes the filtered list of HTTP request and response pairs of U_1 from the filtering component as input and replays each request with the identity of U_2, as illustrated in Figure 4.

The purpose of this is to learn the behavior of the application when URLs that were found during crawling with U_1 and that were not filtered out during the previous step are accessed by U_2. If successful access is possible, then it could be that an access control vulnerability is associated with the

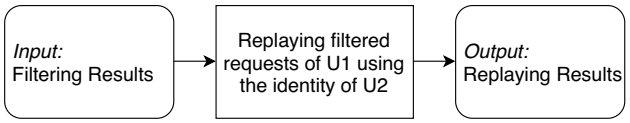

Fig. 4 Replaying component

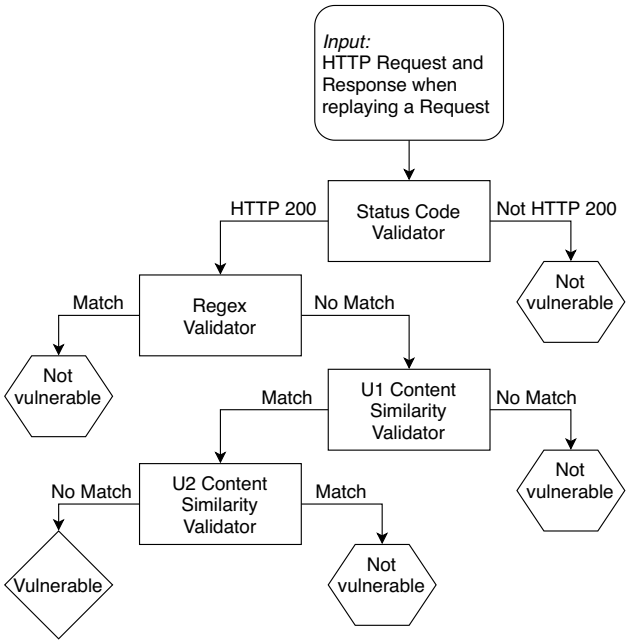

Fig. 5 Validators used to validate the replaying results

corresponding request, but this first has to be verified by the following validating component. The output of the replaying component are the replaying results, i.e., a list of HTTP requests that were replayed using U_2 and the resulting HTTP responses received from the application.

Validating

The final component, the validating component, takes the results from the replaying component and determines, for each HTTP request that was made during replaying, whether an access control vulnerability was detected. To do this, the validating component uses a series of validators, as illustrated in Fig. 5.

First, the *status code validator* checks whether the HTTP response status code indicates successful access. Everything except HTTP 200 status codes is treated as an access denied decision, which means that no vulnerability is detected. In most web applications, the application replies to GET requests that it can handle with an HTTP 200 status code. The other status codes are used to signal a redirection (codes starting with 3, i.e., 3xx), a client error (4xx), or a server error (5xx). If an application strictly follows these status code conventions, permitted and denied access decisions could be determined from the status code alone. In such a scenario, a permitted access would result in a 200 status code and all other status codes would indicate denied access. In practice, however, many applications do not follow these conventions. For example, an application could always reply to a request with a 200 status code and show the user a custom error page indicating the result of a denied access. Badly formed requests, internal server errors or expired user sessions would, however, still lead to status codes different from 200. To summarize, anything other than a 200 status code is considered as an access denied decision, but a 200 status code does not directly identify a permitted access and needs to be analyzed further.

All responses that have a 200 status code are next analyzed by the *regex validator*. This validator looks for specific string patterns inside the response body to decide about permitted and denied access, e.g., custom access denied error messages that appear when an illegitimate access is attempted. If there is a match, this is considered as an access denied decision and consequently, no vulnerability is flagged. Note that this validator must be configured manually and is, therefore, not used in the default configuration.

The third validator, the U_1 content similarity validator, checks whether the content of the response of the replayed request is similar to the response of the original request (details about the mechanism to compare contents will follow in "Content similarity"). If the contents are sufficiently similar, the request is kept as a potential vulnerability, otherwise no vulnerability is flagged. The reason for using this validator is because even if U_2 can successfully access a resource that was found when crawling with U_1, it does not always indicate a vulnerability. For instance, assume U_1 and U_2 are two sellers in an e-shop that offer products. As sellers, both can access a resource that lists their own products, but let us assume that the resource was only found when crawling with U_1. During replaying, U_2 can successfully access the resource and without this validator, this would wrongly be flagged as a vulnerability. Using this validator, however, no vulnerability would be reported because the two sellers offer different products so the contents of the responses are very likely not similar enough for the validator to indicate a vulnerability. As another example, assume that during crawling with U_1, a list of documents is received that can be downloaded by U_1. U_2 does not have the rights to download documents so the URL to get the list was not found when crawling with U_2. But now assume that the application is implemented in a way that it simply returns an empty list during replaying, when it is attempted to access the list using U_2. From an access control point of view, this corresponds to correct behavior. In this case, if this validator were not used, a vulnerability would be reported, which would again be wrong. But using the validator, it is recognized that the

contents are significantly different, and consequently, no vulnerability is reported.

The fourth and final validator is the U_2 *content similarity validator*. This validator was not used in the original paper [1], but based on further analyses in the context of additional evaluations, we gained new insight which led us to the conclusion that this validator improves the overall solution. The validator compares the content received when replaying a request with all contents that were received in responses when crawling with U_2, using the same approach as in the previous validator. If the similarity is sufficiently high for any one of the comparisons, the request is not flagged as a vulnerability. Otherwise, if no significant similarity can be determined, the request is marked as vulnerable. The rationale for this validator is that if U_2 can access basically the same content as U_1 when replaying a request of U_1, but if U_2 could also access this content during crawling, then U_2 apparently has legitimate access to this content. Therefore, the corresponding request should not be flagged as vulnerable. This validator helps significantly to reduce the number of false positives that would be reported if the output of the third validator were used as the final verdict. For instance, assume that U_1 and U_2 use different URLs during crawling to basically access the same content, either because the URL paths indeed differ, because they use different default query parameters (e.g., for sorting), or because the URL includes an anti CSRF token. In all these cases, assuming that U_2 can indeed access the same content also via the corresponding URL found during crawling with U_1, this would result in a false positive after the third validation step. The fourth step, however, will recognize that the content could also be accessed when crawling with U_2 and identify the request correctly as not vulnerable. Another positive side effect of this final validation step is that it is also very efficient at removing requests that point to standard pages, such as custom error pages. As discussed in "Filtering", this was a feature of the originally used content similarity filter that we removed due to other problems. With the fourth validation step, this desired "filtering effect" is brought back, but without the negative side effects of the original filter.

The output of the validating component is a list of requests that correspond to access control vulnerabilities, i.e., resources of U_1 that can illegitimately be accessed by U_2.

Content Similarity

The third and fourth validators used by the *validating component* (see "Validating") determine the similarity of contents of HTTP responses.

There are different approaches to compare such contents for similarity. One option is to directly compare the contents for equality. While fast and simple, this has the major drawback that it cannot detect even the smallest differences between two contents. In practice, when sending the same request twice, there is often no guarantee that the responses are exactly the same because of, e.g., dynamic elements such as server-generated CSRF tokens or timestamps. Therefore, a better approach is required to cope with such differences.

The original paper [1] used fuzzy hashing to solve this. Instead of comparing two contents directly, fuzzy hashes of each content are computed and compared for similarity. To do this, two different fuzzy hashing algorithms were used, ssdeep [9] and tlsh [10]. These algorithms were chosen because tlsh has been proven to perform well when comparing HTML documents [11] while ssdeep is one of the best known and most widely used fuzzy hashing algorithms.

In the meantime, further analysis led us to the conclusion that using fuzzy hashing is not the best way to determine the similarity of contents, for several reasons:

- Even if two contents differ significantly when viewing them in the browser, fuzzy hashing may classify them as similar. For instance, in the case of an e-shop application, two different products when viewed in the browser may look very differently. However, as the entire structure of the underlying HTML documents that define the page layout are the same in both cases, it may easily be that 95% of the two HTML documents are exactly the same as the content visible in the browser is only a small portion of the entire document. As a result of this, using fuzzy hashing, it is likely the two contents will be classified as quite similar.
- Web pages often contain standard headers, footers and navigation areas that typically do not contain sensitive information and that are usually present on all or most web pages within a specific web application. This further distorts determining the similarity of HTML documents by making them more similar than they would be if the focus were only on the relevant content.
- Fuzzy hashing is not robust to sorting. Assume that the two contents to be compared are basically the same web page with the same list of users, but the sorting of the users is different. In this case, although the two contents are obviously the same, fuzzy hashing likely classifies them as quite differently.
- Fuzzy hashing is optimized to work well with significant amounts of data. For instance, ssdeep returns a warning when used with less than 4'096 bytes of data [12]. With relatively big HTML document, this is usually not a problem. However, JSON responses often contain relatively small amounts of data and as a result of this, fuzzy hashing has sometimes difficulties to cope with JSON data.

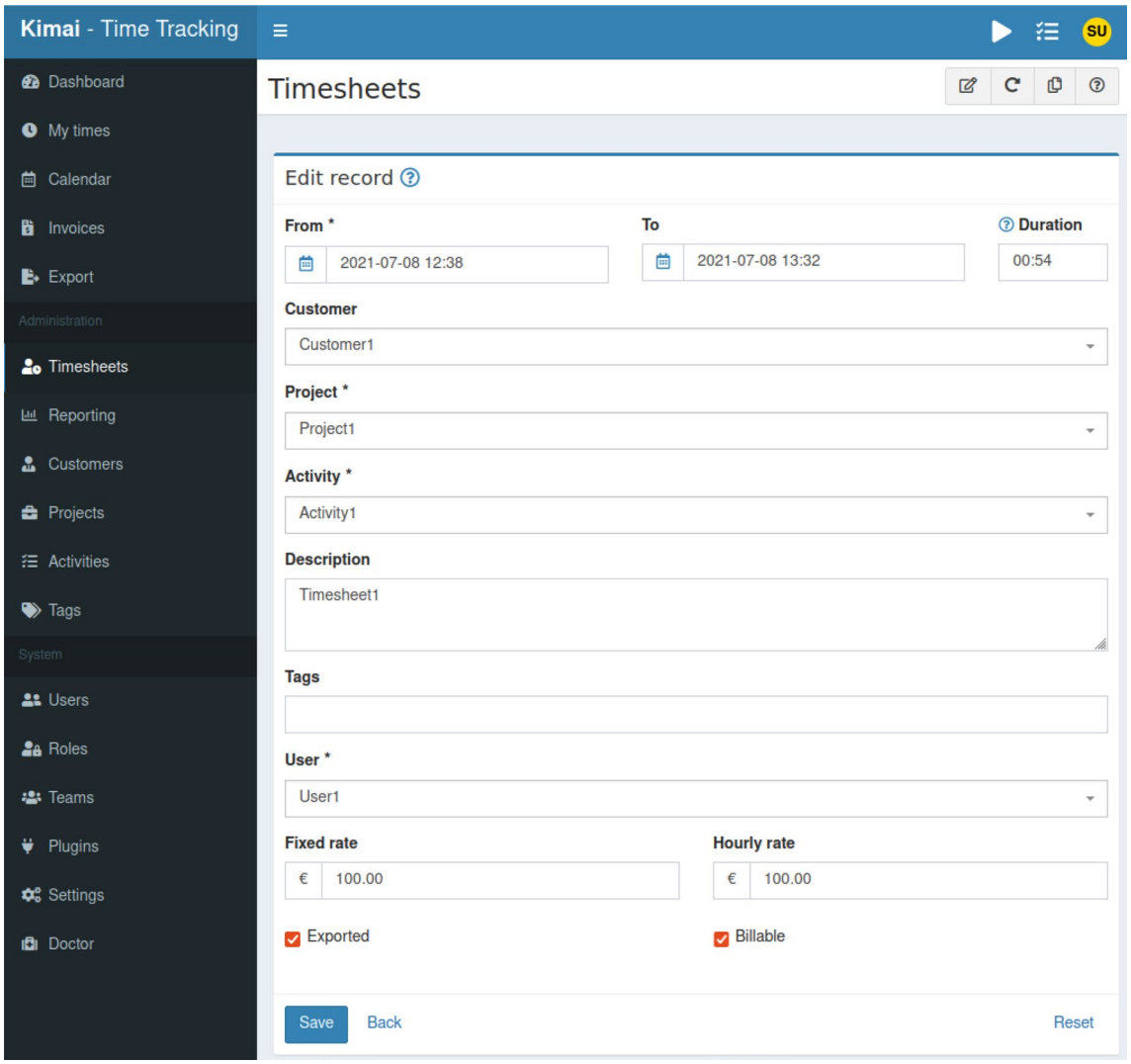

Fig. 6 Web page of Kimai as viewed in the browser

Due to these limitations, we developed a different approach to compare the similarity of two contents. In the case of HTML documents, this is done as follows:

1. Completely remove all aside, footer, header, link, meta, nav and script tags. The contents of these tags are usually either not visible, contain standard headers and footers at the top and bottom of the web page, or mainly serve to navigate the web application. Optionally, it is also possible to configure additional tags that should be removed (see "Configuration example").
2. Of the remaining document, extract all textual elements that are visible in the browser, such as table headings, table content, link texts, button texts, labels and values of input fields.

To illustrate this, Fig. 6 shows a page of the Kimai web application that will be discussed in more detail in "Test application 3: Misago".

After processing the corresponding HTML document (that has a size of 34'047 bytes) as described above, 39 textual elements as illustrated in Fig. 7 remain. In mathematical terms, this is a multiset of textual elements as elements may be present multiple times in an HTML document.

By comparing the textual elements with the visible web page in Fig. 6, one can easily see that they mostly correspond to the elements that are visible in the right part of the web page, which is the actual main content of the page. The side bar on the left used for navigation has been filtered out in the first step of the approach described above. There are some textual elements included in Fig. 7 that are not directly

Fig. 7 Textual elements extracted from the Kimai web page in Fig. 6

```
{Kimai - Time Tracking, Timesheets, Edit record, From, 2021-07-08
 12:38, To, 2021-07-08 13:32, Duration, 00:54, Customer, Customer1,
 Customer2, Project, Project1, Activity, Activity1, Description,
 Timesheet1, Tags, User, Superadmin, Superadmin2, Teamlead1, Teamlead2,
 User1, User2, Fixed rate, € , 100.00, Hourly rate, € , 100.00, Exported,
 1, Billable, 1, Save, Back, Reset}
```

visible in Fig. 6, e.g., *Superadmin* and *Teamlead1*, because they are hidden behind the drop-down lists on the web page.

To determine the content similarity of two HTML documents, the two multisets that contain their textual elements are used. It is calculated by dividing the size of the intersection of the two multisets by the size of the larger of the two multisets. Expressed as a formula, the content similarity CS of the two sets TE_1 and TE_2 that contain the textual elements of two HTML documents is expressed as follows:

$$CS = 100 \cdot \frac{|TE_1 \cap TE_2|}{max(|TE_1|, |TE_2|)}. \quad (1)$$

This results in a value of CS between 0 and 100. If the two multisets are the same, then $CS = 100$. If they are disjoint, then $CS = 0$. If $TE_1 = \{a, a, b, c, d, e\}$ and $TE_2 = \{a, a, b, d, e, f, g, g\}$, i.e., four textual elements are present in both multisets and the larger of the two multisets has eight elements, then $CS = 50$.

This new approach to determine the content similarity solves the issues that existed with the previously used method. First of all, the focus is now on the actual visible parts of the web page while irrelevant parts of the HTML documents such as tags and the standard content of headers, footers and navigation areas are ignored. Second, the new approach is insensitive to any sorting order of the relevant elements on a web page as calculating the content similarity according to (1) only considers the textual elements per se and not any ordering. And third, unlike fuzzy hashing, the method works well with any size of HTML documents.

In the case of JSON data, the data are processed differently. Just like with HTML documents, the goal is to have a multiset of relevant textual elements that are included in the JSON data. In general, JSON data contain only or mostly relevant data and in particular, they usually do not contain formatting information such as HTML tag. Therefore, no data is removed from the JSON data. To transform JSON data into a multiset of textual elements, the JSON data structure is flattened. To illustrate this, Fig. 8 shows a simple JSON data structure received from the Misago web application that will be discussed in more detail in "Test application 7: Kimai".

As one can see in Fig. 8, JSON data can include multiple key-value pairs that have the same key. For instance, the key *id* shows up three times, but each time, it identifies a different object. The topmost *id* identifies a specific message within Misago while the other two *id*s identify two different likes this message received. Therefore, when extracting textual elements from JSON data, one should not only use the innermost key-value pairs, but for each pair, one should also consider the key(s) of the surrounding JSON object(s) to clearly separate the textual elements. For instance, the textual element resulting from the key-value pair *"id": 4* should also include the key of the surrounding JSON object, which is *last_likes*.

Figure 9 shows the resulting multiset when extracting the textual elements from the JSON data in Fig. 8, i.e., when flattening it.

As one can see, all extracted key-value pairs also contain the key(s) of the surrounding JSON element(s) (if there is/are any). For instance, the key-value pair *"id": 4* within the JSON object with key *last_likes* results in the textual element *last_likes_id:4*. Note that double quote characters in the JSON data are omitted when extracting the textual elements.

To determine the content similarity of two JSON data structures, the two multisets that contain their textual elements are used in formula (1), i.e., exactly the same formula as with HTML documents is used.

If the content contains something else than an HTML document or JSON data, e.g., a binary PDF document or an image, no textual elements are extracted and the entire

```
{
  "id": 6,
  "likes": 2,
  "last_likes": [
    {
      "id": 4,
      "username": "member1"
    },
    {
      "id": 2,
      "username": "moderator1"
    }
  ],
  "is_liked": true,
  "detail": [
    "ok"
  ]
}
```

Fig. 8 JSON data received from Misago

Fig. 9 Textual elements extracted from the Misago JSON object in Fig. 8

```
{id:6, likes:2, last_likes_id:4, last_likes_username:member1,
 last_likes_id:2, last_likes_username:moderator1, is_liked:True,
 detail:ok}
```

Fig. 10 Configuration example

```
target:
  target_url: https://www.site.com
  target_domain: site.com
auth:
  auth_user_1: Cookie cookie-value-user-1
  auth_user_2: Cookie cookie-value-user-2
  username_user_1: userA
  username_user_2: userB
options:
  do_not_call_pages: logout,logoff,log-out
  static_content_extensions: js,css,img,jpg,png,svg,gif
  standard_pages: about.php,credits.php
  regex_to_match:
  tags_to_remove:
  threshold_validator_3: 80
  threshold_validator_4: 95
```

content is treated as "one big textual element". Consequently, when comparing such documents, the determined similarity can only be 100 if the contents are equal or 0 if they are different.

One final question remains: What threshold should be used to decide whether two contents are considered similar? This has been determined by experimentation during the evaluation (see "Evaluation"), which led to the following choice of similarity thresholds:

- The third validator of the validating component (see "Validating") uses a threshold of 80. This means that if there is an overlap of 80% or more of the textual elements extracted from the two contents that are compared, then they are considered similar.
- The fourth validator of the validating component (see "Validating") uses a threshold of 95, which means an overlap of at least 95% is required for similarity. The reason for using a higher threshold compared to the third component is to make sure that potential vulnerabilities identified after the third component are only filtered out if there is an almost complete overlap of the content received during replaying with content received when crawling with U_2.

Testing of Multiple Users and Roles

As mentioned in "Overall work", the workflow in Fig. 1 can determine vulnerabilities where U_2 can illegitimately access resources of U_1. However, there are often more than two users and roles that should be considered. To support this, the workflow is simply used repeatedly for each pair of users or roles that should be tested, where U_2 can also be the anonymous (unauthenticated) user. This can easily be automated by writing a wrapper program that repeatedly executes the entire workflows for all user pairs.

For example, if an application provides three user roles *administrator (A)*, *vip user (V)* and *standard user (S)*, and if the *anonymous user (Y)* should also be considered and one wants to find vulnerabilities between each pair of distinct users or roles, then nine runs of the entire workflow would be done based on the pairs *(A,V)*, *(A,S)*, *(A,Y)*, *(V,A)*, *(V,S)*, *(V,Y)*, *(S,A)*, *(S,V)* and *(S,Y)*. Note that if the anonymous user is used for U_2, the entire workflow basically works as described above, but crawling with the unauthenticated user is omitted (as this is already done when crawling with U_2) and consequently, the public content filter is also not used.

Configuration Example

To give an idea about the configuration that is required for the solution to detect vulnerabilities based on two users, Fig. 10 shows the default configuration file.

The first two sections *target* and *auth* are self-explanatory and must be specifically configured. As can be seen, the required configuration effort is small. The third section *options* can be adapted depending on the web application under test, which may improve vulnerability detection. The parameters in this section mean the following:

- *do_not_call_pages*: URLs that the crawlers should not follow, e.g., to avoid logging themselves out.
- *static_content_extensions*: Extensions of static content that are used by the static content filter.

- *standard_pages*: Standard pages that are used by the standard pages filter.
- *regex_to_match*: The regular expression used by the regex validator (empty per default).
- *tags_to_remove*: Additional HTML tags that should be removed from HTML documents by the two content similarity validators. This can be used if, e.g., an application uses non-standard tags for navigation areas (empty per default).
- *threshold_validator_3/threshold_validator_4*: Thresholds for the third and fourth validator of the validating component. The default values 80 and 95 have demonstrated to work well during the evaluation.

Implementation

The described solution was implemented as a fully functional prototype using Python. To capture all requests and responses made by the crawling component, a web proxy was built based on the proxy library *mitmproxy* [13]. The proxy saves all requests and responses plus additional meta data (e.g., which user authentication information was used for the request and which crawler was used) to a database. For the database, *SQLite* [14] is used.

The prototype implementation does not require any special hardware. During development and evaluation, a standard Linux-based server system with 4 CPU cores and 8 GB RAM was used.

Evaluation

To evaluate the prototype and the entire solution approach, a set of test applications is required, each containing at least one access control vulnerability. To create such a set, access control vulnerabilities were added to six available web applications by modifying the application code or access control configurations. In addition, we used one web application with a vulnerability that has been publicly disclosed. In total, the test set consists of seven web applications that represent a diverse set of traditional and modern web technologies and frameworks.

Compared to the original paper [1], the evaluation contains three additional applications. Sections "Test Application 1: Marketplace"–"Test Application 4: WordPress" include the four applications that were already evaluated in the original paper while "Test Application 5: Magento"–"Test Application 7: Kimai" contain three new applications. Also, the focus in the original paper was mainly on a quantitative analysis of the filtering effectiveness of the different stages of the filtering component, the number of vulnerabilities that could be detected, and the number of false positives. While this analysis served well to demonstrate that the overall approach works in principle, it provided limited insight why vulnerabilities were missed in detail or why false positives occurred.

The evaluation in this paper will, therefore, not focus on the aspects that were already discussed in the original paper. Instead, there will be a detailed discussion of why a specific vulnerability could be detected or not, why a specific false positive was reported, and what improvements may further be possible to increase the vulnerability detection performance of the solution. This will provide much deeper insights than the evaluation in the original paper and will significantly help to understand the true possibilities and limitations of the solution.

Unless stated otherwise, the default options as shown in Fig. 10 are used during the evaluation.

Test Application 1: Marketplace

The first application used for evaluation is a very simple web shop application that is used at our university for educational purposes. It is based on Node.js Express and uses a classic architecture where the browser gets complete HTML documents from the server that do not include any JavaScript code. Also, with about ten different resources (URLs), the application is small in scope and, therefore, well suited as a starting point to evaluate the solution. The application has an administrative area where users with the appropriate roles can add and delete products that are offered in the shop and view and delete the purchases that were made in the shop. The two relevant roles in this context are the following:

- Role *Product Manager (P)*: can add and delete products.
- Role *Sales (S)*: can view and delete purchases.

One access control vulnerability (identified as V_1) was added to the application:

- V_1 is located in the Add Product functionality, which should only be accessible by users with the role *P*. Due to the vulnerability, every authenticated user (i.e., also users with role *S*) can access this resource to add a product. The URL to access this resource is as follows[1]: http://www.site.com/admin_product_add

To evaluate whether V_1 can be detected, a user with role *P* is used as U_1 and a user with role *S* is used as U_2 as inputs to the entire workflow and the crawling component (see Figs. 1 and 2). After the filtering component (see Fig. 3),

[1] For simplicity and for all web applications included in the evaluation, we write *http* independent of whether *http* or *https* is used to access the web application and *www.site.com* for the host name.

Table 1 Summary of evaluation results of Marketplace

Vuln.	URL	Detected
V_1	http://www.site.com/admin_product_add	Yes
No false positives		

only the vulnerable resource http://www.site.com/admin_product_add is left and anything else has been filtered out. Next, during the replaying component (see Fig. 4) this URL is replayed with U_2, which—due to the vulnerability—works and creates an HTTP response with status code 200. Finally, during the validating component, it is determined whether this URL is vulnerable or not using the four validators illustrated in Fig. 5:

1. The status code validator sees that the status code of the response is 200, so the next validator is called.
2. The regex validator does nothing, as no regular expression is configured, so the next validator is called.
3. The U_1 content similarity validator compares the HTML document when replaying the URL with the HTML document that was received when crawling with U_1. As both documents are exactly the same, the content similarity value according to (1) results in a value of 100, which is obviously above the similarity threshold of 80. Therefore, the next validator is called.
4. The U_2 content similarity validator compares the HTML document when replaying the URL with all HTML documents that were received when crawling with U_2. As all these documents are very different from the one received during replaying, the maximum similarity value according to (1) is 17, which is clearly below the similarity threshold of 95. As a result of this, the URL http://www.site.com/admin_product_add is correctly identified as a vulnerability.

As only one URL is remaining after the filtering component no further URLs are replayed and consequently, there are no false positives.

Overall, this first evaluation demonstrates that the solution works well with a simple application and—in this case—delivers an optimal result: The solution could identify the vulnerability without reporting any false positives. Table 1 summarizes the evaluation results of the Marketplace application.

Test Application 2: Bludit

Bludit [15] is a Content Management System implemented in PHP that uses a classic architecture where most of the logic is implemented in the backend. It contains almost 200 different URLs, so it is a much bigger application than the first one and therefore better suited to evaluate the effectiveness of the overall approach. It has an extensive privileged area where authenticated user can add, modify and delete content. The roles we are considering here are the following:

- Role *Author (T)*: create, modify and delete own content
- Role *undefined*: all permissions of *T*; in addition create, modify and delete any content of any user, change the appearance of the website, install plugins, manage user accounts, and change numerous settings

In this application, three different access control vulnerabilities V_1–V_3 were implemented:

- V_1 is located in the plugin settings area, which should only be accessible by users with role *A*. Due to this vulnerability, users with role *T* can access the resources where all plugin settings can be viewed and modified. The URL to access such a resource is, e.g., as follows: http://www.site.com/admin/configure-plugin/pluginCategories
- V_2 is similar to V_1 but affects a different resource, namely the page where a user with role *A* can view and change general website settings. The URL to access this resource is as follows: http://www.site.com/admin/settings
- V_3 allows users with role *T* to get access to resources where content of other users with role *T* can be modified. The URL to access such a resource is, e.g., as follows: http://www.site.com/admin/edit-content/content-of-author1

As Bludit uses non-standard tags for the navigation area, the *tags_to_remove* options (see Fig. 10) was set to *'class':'nav flex-column pt-4'*. This allows the *content similarity validators* to correctly remove the navigation areas.

To evaluate whether V_1 can be detected, a user with role *A* is used as U_1 and a user with role *T* is used as U_2. The corresponding URL is found during crawling with U_1, is not filtered out by the filtering component, and is therefore replayed. The content similarity determined by the U_1 content similarity validator is 100 (and, therefore, ≥ 80), so the textual elements extracted from the HTML document during replaying are exactly the same as the ones extracted from the HTML document received when crawling with U_1. All content similarities determined by the U_2 content similarity validator are 25 or lower (and therefore < 95). Consequently, V_1 is correctly detected.

This vulnerability serves well to illustrate why removing navigation areas is important before the content similarity is determined. Figure 11 shows the web pages received when accessing the vulnerable URL http://www.site.com/admin/

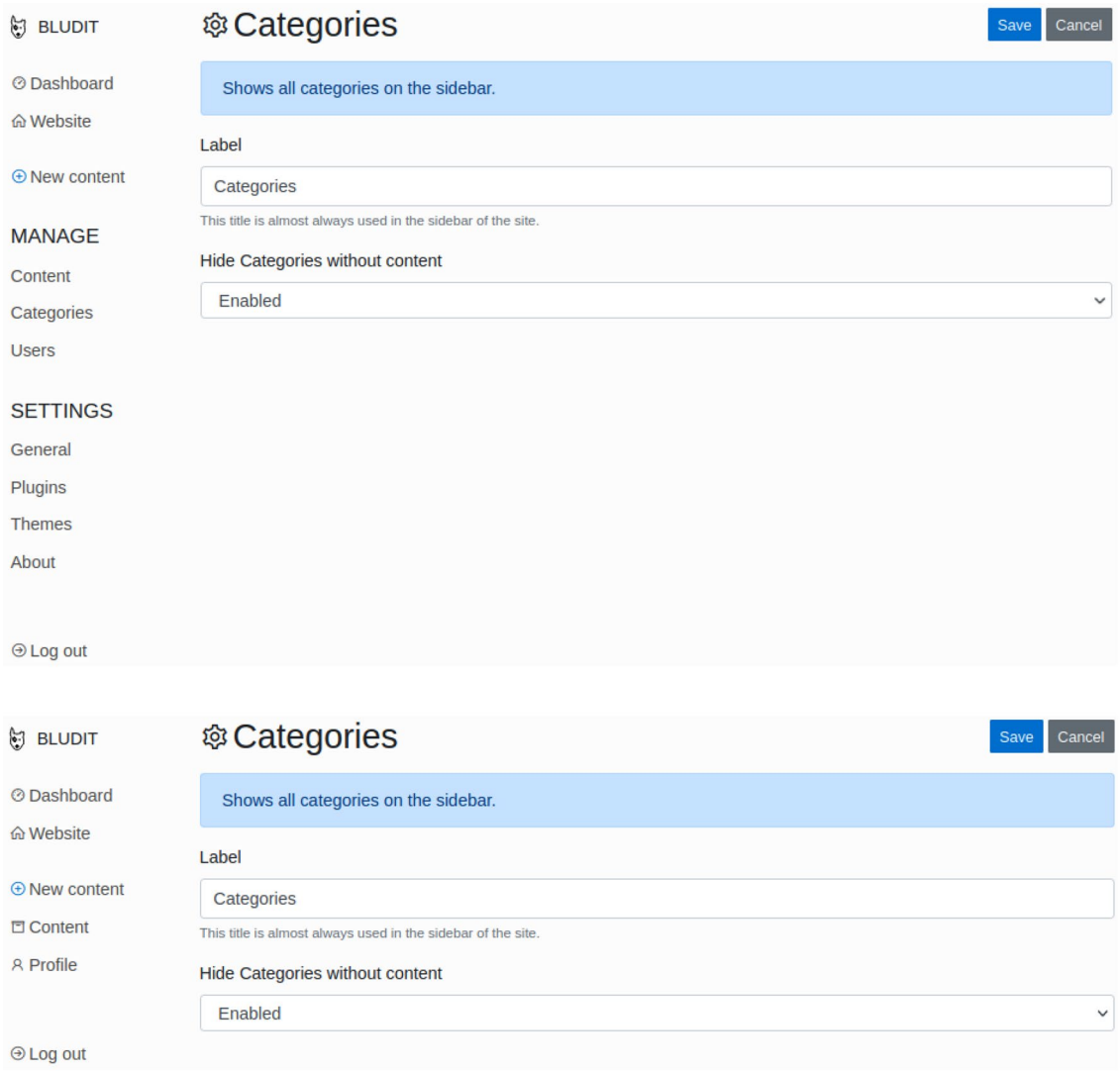

Fig. 11 Response by Bludit when accessing http://www.site.com/admin/configure-plugin/pluginCategories with role Administrator (A) (top) and Author (T) (bottom)

configure-plugin/pluginCategories with users with roles *A* and *T*.

As can easily be seen in Fig. 11, the main content is the same in both cases. However, the navigation areas on the left looks quite different in the two cases. The reason is that although the resource is vulnerable, the navigation area always corresponds to the one of the current role. If the navigation area were not removed before extracting the textual elements from the HTML documents, the resulting multisets of textual elements would be quite different from each other. In that case, also because the main content only contains relatively few textual elements, the resulting content similarity value determined by the U_1 content similarity validator would only be about 60 and as a result of this, the URL would wrongly be classified as not vulnerable.

To check whether V_2 can be detected, the same users as in the case of V_1 are used. The result of this is that V_2 cannot be detected because the vulnerable URL is filtered out by the other user content filter. The reason for this is a strange application design choice: for the privileged area, the developers chose to implement two different navigation menus, one for devices with a small and one for devices with a bigger screen. While the navigation links are correctly implemented in one of these menus, the other one exposes all navigation links, even those a user is not permitted to visit. Therefore, during crawling, the vulnerable URL can also be found by U_2 (i.e., by a user with role *T*) and is, therefore, filtered out. This is a case where the fundamental assumption (see beginning of "Solution approach") is not valid and, therefore, the vulnerability cannot be detected by design.

Table 2 Summary of evaluation results of Bludit

Vuln.	URL	Detected
V_1	http://www.site.com/admin/configure-plugin/pluginCategories	Yes
V_2	http://www.site.com/admin/settings	No
V_3	http://www.site.com/admin/edit-content/content-of-author1	Yes
No false positives		

To evaluate whether V_3 can be detected, two different users with role T are used as U_1 and U_2. The corresponding URL is found during crawling with U_1, is not filtered out by the filtering component, and is, therefore, replayed. The content similarity determined by the U_1 content similarity validator is 100 ($\geq$ 80) and content similarities determined by the U_2 content similarity validator are all below 95. Therefore, V_3 is correctly detected.

No false positives are reported. Note that in the original paper, two false positives were mentioned. They occurred with several user combinations and affected the URLs http://www.site.com/admin/edit-user/user1 and http://www.site.com/admin/user-password/user1, where user1 can be any existing username. These URLs point to the pages to change user settings and user passwords. During crawling, U_1 finds the URLs with its own username (e.g., *user1*) and U_2 also finds the URLs with its own username (e.g., *user2*). Therefore, during replaying, U_2 accesses the URLs with the username of U_1, i.e., user1. However, the application is implemented in a way that when accessing these URLs with the username of another user, the web application returns an HTML document as if the own username were used. This means that if U_2 requests http://www.site.com/admin/edit-user/user1, the web application delivers exactly the same content as when accessing http://www.site.com/admin/edit-user/user2. As a result of this, the content similarity determined by the U_1 content similarity validator is 100. As the fourth and final validator of the validating component was not used in the original paper, false positives were reported. With the newly introduced U_2 content similarity validator, however, it is recognized that the content received during replaying was already received when crawling with U_2, and therefore, the URL is correctly classified as not vulnerable. This nicely illustrates the usefulness of the newly added fourth validator.

This second evaluation demonstrates that the solution can also handle web applications that contain a significant number of URLs, that it is important to remove navigation areas before the content similarity is determined, and that the new fourth validator can be helpful to prevent false positives. However, it can also be seen that false negatives (V_2 in this case) may occur if the application violates the fundamental assumptions that users can only crawl content they can legitimately access. Table 2 summarizes the evaluation results of Bludit.

Test Application 3: Misago

Misago [16] provides a web forum solution based on Django and React. It uses a modern web application architecture, meaning that a lot of functionality is implemented client-side using JavaScript code while the backend provides a REST API. For such an application, the Puppeteer-based crawler is paramount to discover the API endpoints. Aside from being a good test application to evaluate the crawling component of the solution, it is also a good case study to find out whether the content similarity validators can handle JSON data that are mostly used by modern web application architectures to exchange data between front- and backend. With more than 800 URLs detected during crawling, the application is significantly bigger than the previous ones. The following three roles are considered in this evaluation:

- Role *Member (M)*: create own threads and participate in the threads of others
- Role *Moderator (O)*: all permissions of *M*; in addition moderation functions such as removing posts and banning members
- Role *Administrator (A)*: all permissions of *M*; in addition access to the administrator interface to do forum management, user management, etc.

In this application, three different access control vulnerabilities V_1—V_3 were implemented:

- V_1 is located in the administrator interface, specifically in the resource that lets a user with role *A* view and manage users of a forum. The vulnerability allows users with role *O* to view the complete list of users of the forum. The URL to access this resource is as follows: http://www.site.com/admincp/users
- V_2 can be found in a feature called *Private Thread*, which allows users to start a private conversation with anyone they invite to the thread. The vulnerability allows any authenticated user to view the content of any private thread. This vulnerability manifests itself both in the navigation URL and in the REST API in the backend, which serves the content of the thread. The navigation URL to access such resources is, e.g., http://www.site.com/p/test-thread/5 and the corresponding URL to access the API is, e.g., http://www.site.com/api/private-threads/5/posts/9/post_editor.
- V_3 is present in the user profile. When trying to change details of the own profile, a request to an API endpoint is first made to gather the current user information. The

vulnerability allows all authenticated users to get the details of any other user, except of users with role *A*. This vulnerability is only present in the REST API, but not in the navigation URL and is therefore a good test case to see whether the solution can detect access control vulnerabilities that can only be reached directly via JSON-based REST APIs. The URL to access the API is, e.g., http://www.site.com/api/users/4/edit-details.

To evaluate whether V_1 and V_2 can be detected, a user with role *A* is used as U_1 and a user with role *O* is used as U_2. In the case of V_1, the vulnerable URL is found during crawling with U_1, is not filtered out by the filtering component, and is therefore replayed. The content similarity determined by the U_1 content similarity validator is 97 ($\geq$ 80). The small difference is due to the fact that the HTML documents include some information about the server-side processing time in ms, which is slightly different in both cases. This shows that it is important to not require a perfect match when comparing data received from the application because as even seemingly identical responses often have small differences such as statistical information, the username of the current user, etc. All content similarities determined by the U_2 content similarity validator are 25 or lower ($<$ 95). Consequently, V_1 is correctly detected.

In the case of V_2, both URLs are found during crawling with U_1, are not filtered out by the filtering component, and are therefore replayed. With URL http://www.site.com/p/test-thread/5, the content similarity determined by the U_1 content similarity validator is 93 ($\geq$ 80) and all content similarities determined by the U_2 content similarity validator are 61 or lower ($<$ 95). Consequently, this URL is correctly identified as vulnerable. With URL http://www.site.com/api/private-threads/5/posts/9/post_editor, the content similarity determined by the U_1 content similarity validator is 100 ($\geq$ 80) and all content similarities determined by the U_2 content similarity validator are 57 or lower ($<$ 95). Therefore, this URL is also correctly identified as vulnerable.

Finally, for V_3, two different users with role *M* are used as U_1 and U_2. The corresponding URL is found during crawling with U_1, is not filtered out by the filtering component, and is therefore replayed. The content similarity determined by the U_1 content similarity validator is 100 ($\geq$ 80) and content similarities determined by the U_2 content similarity validator are at most 82 ($<$ 95). Therefore, V_3 is correctly detected.

In the original paper, testing Misago produced several false positives. Many of them could be removed because of the adaptations we made to the overall solution, in particular by adding the U_2 content similarity validator. Others could be removed using better test data. For instance, in Misago, every user can access the profiles of all other users. However, a user only finds a link to the profile of another user if the other user has already posted a message where the first user has access to. This has been taken into account during the evaluation for this paper, i.e., we made sure that every user has posted at least one public message. Another general recommendation with respect to test data is to make sure the data are realistic and that data used in different instances of the same type (e.g., different messages, different user profiles, etc.) is truly different. To illustrate this, let's revisit V_3. Initially, V_3 could not be detected as the U_2 content similarity validator determined a similarity value of 98 when comparing the JSON data received during replaying with the JSON data U_1 received when accessing the own profile during crawling. The reason was that the profiles of the two users only differed in one value of the JSON data, namely in the actual usernames that were used (member1 and member2). The other about 50 values included in the JSON data were exactly the same in both cases and therefore, the similarity of the two JSON data structures was close to 100. By configuring the user profiles so they are more clearly distinguishable (e.g., not only different user names, but also different real names, different Skype IDs, different websites, etc.), the content similarity gets lower (to 82) and the problem could be fixed.

One false positive, identified as FP_1, is still detected when using a user with role *A* for U_1 and a user with role *O* for U_2. The URL http://www.size.com/api/username-changes/?user=1 is crawled by U_1 but not by U_2. The URL returns JSON data containing the username changes done by U_1. During replaying, U_2 could access the URL and got exactly the same JSON data as U_1 during crawling and consequently, the URL is flagged as a vulnerability. One could argue that this is indeed a vulnerability as U_2 cannot issue this request by navigating the application. However, U_2 can get the same information also via http://www.site.com/u/admin/1/username-history/, which is found by U_2 during crawling. This means U_2 cannot learn anything new by accessing the reported URL and, therefore, we identify this as a false positive—although it certainly is a borderline case.

This third evaluation demonstrates that the solution can also handle web applications that follow a modern architecture with a REST API in the backend and lots of JavaScript code running in the browser. It also shows that by carefully selecting the test data, the vulnerability detection performance can be improved and the number of false positives can be reduced. Table 3 summarizes the evaluation results of Misago.

Test Application 4: WordPress

The fourth test application is a standard WordPress site (based on PHP) with a vulnerable version of the plugin Job Manager [17]. The plugin allows to list job offers and lets users of the site submit job applications, and contains a published access control vulnerability (CVE-2015-6668

Table 3 Summary of evaluation results of Misago

Vuln.	URL	Detected
V_1	http://www.site.com/admincp/users	Yes
V_2	Navigation: http://www.site.com/p/test-thread/5	Yes
	API: http://www.site.com/api/private-threads/5/posts/9/post_editor	Yes
V_3	http://www.site.com/api/users/4/edit-details	Yes
False positives	URL	
FP_1 (borderline)	http://www.size.com/api/username-changes/?user=1	

Table 4 Summary of evaluation results of WordPress

Vuln	URL	Detected
V_1	http://www.site.com/wp-content/uploads/2021/08/cv.pdf	Yes
No false positives		

[18]). About 600 URLs are found during crawling and only one role is considered in this evaluation:

- Role *Author (A)*: create job applications and view and remove own applications

The access control vulnerability has the following behavior:

- V_1 exists because the plugin does not enforce any access control checks on media files, e.g., PDF documents, that are submitted as part of an application. Therefore, the files can, e.g., be accessed by other users with role *A*. The URL to access such a resource is, e.g., as follows: http://www.site.com/wp-content/uploads/2021/08/cv.pdf.

Two resources were added to the *do_not_call_pages* options (Fig. 10): *wp-login.php* and *load-scripts.php*. The first is required to prevent that the crawling component is caught in an endless crawling loop. Such problems can easily be detected by inspecting the logs in case crawling takes a long time. The second is used by WordPress to load JavaScript code (so it is comparable to accessing JavaScript file directly) and is not relevant in the context of access control vulnerabilities.

To evaluate whether V_1 can be found, two different users with role *A* are used. The first one is used as U_1 and has created a job application with a PDF file reachable at http://www.site.com/wp-content/uploads/2021/08/cv.pdf. The second user is used as U_2. The URL to the PDF file is found during crawling with U_1, is not filtered out by the filtering component, and is therefore replayed, which results in receiving exactly the same PDF document. Consequently, the content similarity determined by the U_1 content similarity validator is 100 and all content similarities determined by the U_2 content similarity validator are 0. Therefore, V_1 is correctly detected. No false positives are reported.

Overall, this fourth evaluation demonstrates that the solution can also find real vulnerabilities. Table 4 summarizes the evaluation results of the WordPress application.

Test Application 5: Magento

Magento [19] is a PHP-based e-commerce platform. It uses a modern architecture, where lots of functionality is implemented client-side using JavaScript code, and a REST API in the backend. The crawling component finds more than 2'000 URLs, which makes Magento the largest web application in the evaluation. Two roles are used in this evaluation:

- Role *Administrator (A)*: full access to administrative functions
- Role *Sales (S)*: access to sales-related functions such as customer management, product management, order management, etc.

Two access control vulnerabilities V_1 and V_2 were added to Magento:

- V_1 allows users with role *S* to view pending product reviews. The URL to access this resource is as follows: http://www.site.com/admin/review/product/pending/
- V_2 allows users with role *S* to view the page where product reviews can be edited. The URL to access such a resource is, e.g., as follows: http://www.site.com/admin/review/product/edit/id/1/

To evaluate whether V_1 and V_2 can be detected, a user with role *A* is used as U_1 and a user with role *S* is used as U_2. In both cases, the vulnerable URL is found during crawling

Table 5 Summary of evaluation results of Magento

Vuln.	URL	Detected
V_1	http://www.site.com/admin/review/product/pending/	Yes
V_2	http://www.site.com/admin/review/product/edit/id/1/	Yes
False positives	URL	
FP_1	http://www.site.com/admin/yotpo_yotpo/report/reviews/	
FP_2	http://www.site.com/admin/catalog/product/edit/id/3/	

with U_1, is not filtered out by the filtering component, and is therefore replayed. With V_1, the content similarity determined by the U_1 content similarity validator is 100 ($\geq$ 80) and all content similarities determined by the U_2 content similarity validator are 37 or lower ($<$ 95). With V_2, the similarity values are 100 and 8, respectively. Consequently, V_1 and V_2 are correctly detected.

Two false positives, identified as FP_1 and FP_2, are reported. The first one affects the URL http://www.site.com/admin/yotpo_yotpo/report/reviews/, which can only be crawled by U_1 but not U_2. The received HTML document provides instructions how a third-party marketing platform can be connected. It is obviously not access control-relevant and must, therefore, be classified as a false positive. This is again a case where the fundamental assumption (see beginning of Section 3) is not valid and correspondingly, FP_1 occurred by design. In practice, it would easily be detected after the first run and, based on this, one would add it to the *standard_pages* options in the configuration file (see Figure 10).

FP_2 affects the URL http://www.site.com/admin/catalog/product/edit/id/3/ (and further IDs, but technically they are all the same false positive). Although manually navigating the application shows that these URLs can be found by U_2, they are not detected by the crawling component. A more detailed analysis shows that the corresponding links are included client-side by JavaScript code. However, they are included as $< a >$ tags and not as $<$ *button* $>$ tags and as the Puppeteer-based crawler in its current state cannot cope with such dynamically inserted $< a >$ tags, the URL is not found. This false positive is, therefore, a technical limitation of the current solution and it is likely that it can be fixed by adapting the Puppeteer-based crawler.

This fifth evaluation demonstrates that the solution can also handle very large web applications with thousands of URLs. Reporting just two false positives corresponds to a very small fraction of the total number of URLs, which shows the fundamental assumption on which the entire solution is based (see "Solution approach") seems to be sound also in a large application such as Magento. However, the results also show that there's still room for improvement, as FP_2 was reported due to shortcomings in the crawling component and not due to limitations in the solution approach. Table 5 summarizes the evaluation results of Misago.

Test Application 6: Mattermost

Mattermost [20] is chat application based on a modern architecture and uses Go for the backend and React for the frontend. Being a pure single page application, the Scrapy-based crawler finds only the base HTML document and all other resources can only be detected by the Puppeteer-based crawler. Overall, about 500 URLs were detected. One role is used in this evaluation:

- *Role Member (M)*: access to functions such as reading and writing messages, creating private channels (that are accessible only by members of the channel), and joining public channels

One access control vulnerability V_1 was added to Mattermost:

- V_1 allows users with role M to view image previews of messages that were posted in private channels where the user is not a member. The response contains the image as binary data. The URL to access such a resource is, e.g., as follows: http://www.site.com/api/v4/files/1kq1ej8x1fg6fyex9ykf6hmyyr/preview

To evaluate whether V_1 can be discovered, two users with role M are used for U_1 and U_2. U_1 has posted a message including an image in a private channel and U_2 is not a member of this channel. The vulnerable URL is found during crawling with U_1, is not filtered out by the filtering component, and is, therefore, replayed and delivers the same binary image data. As the images are equal, the content similarity determined by the U_1 content similarity validator is 100 and all content similarities determined by the U_2 content similarity validator are 0. Consequently, V_1 is correctly detected.

One false positive, identified as FP_1 is reported. It concerns the URL http://www.site.com/api/v4/users/w1nhdw38it8exxgbo81turafjo/status. Such a URL is called when accessing the main page to get the own status, where the

Table 6 Summary of evaluation results of Mattermost

Vuln.	URL	Detected
V_1	http://www.site.com/api/v4/files/1kq1ej8x1fg6fyex9ykf6hmyyr/preview	Yes
False positives	URL	
FP_1	http://www.site.com/api/v4/users/w1nhdw38it8exxgbo81turafjo/status	

the URL part *w1nh ...afjo* is user-specific. The response is a small JSON data structure that contains information such as the status of the user (e.g., online/offline) and the timestamp of the last activity. As every user crawls its own URL, U_2 crawls a different URL than U_1, but U_2 can successfully access the URL crawled by U_1 during replaying and also gets exactly the same content. Technically, this can be classified as a vulnerability. However, it seems that this is implemented intentionally in this way as the application also includes functionality where the status of other users can be accessed by other means. Also, the received information is not really sensitive. Consequently, this is classified as a false positive.

Similar to Misago (see "Test Application 3: Misago"), this sixth evaluation confirms that thanks to the Puppeteer-based crawler, the solution can cope well with web applications that follow a modern architecture, in this case a pure single page application approach. Table 6 summarizes the evaluation results of Mattermost.

Test Application 7: Kimai

Kimai [21] is a time-tracking application based on PHP. It uses a modern architecture and about 400 URLs are detected during crawling. Two roles are used in this evaluation:

- Role *Superadmin (S)*: can manage everything in Kimai, including users, their timesheets, plugins, and system configuration
- Role *Teamlead (T)*: can manage teams and has access to all timesheets within these teams

Two access control vulnerability V_1 and V_2 were added to Kimai:

- V_1 allows users with role T access to read the preferences of all users, including those of users with role S. Normally, only users with role S should have access to all profiles. The URL to access such a resource is, e.g., as follows: http://www.site.com/en/profile/superadmin/prefs
- V_2 allows users with role T access the page to change the password of other users. The URL to access such a resource is, e.g., as follows: http://www.site.com/en/profile/superadmin/password

Kimai requires some additional configuration options to make sure it can be tested with good results. First of all, some resources must be added to the do_not_call_pages options (see Fig. 10): working-time, week-by-user, month_by_user, year_by_user, weekly_users_list, monthly_users_list and yearly_users_list. Calls to these resources are provided to the users in the form of backward and forward arrow buttons to navigate the timesheets by week, month and year, and adding them to the do_not_call_pages prevents that the crawling component ends up in an endless crawling loop. As already explained in the context of the WordPress application (see "Test Application 4: WordPress"), such loops can usually easily be detected by inspecting the logs if crawling takes a long time. The second configuration option is required as Kimai uses non-standard tags for the navigation areas. Therefore, the *tags_to_remove* option was set to *'class':'main-sidebar','class':'box-tools','class':'hidden-xs'*. This allows the content similarity validators to correctly remove the navigation areas.

To evaluate whether V_1 and V_2 can be detected, a user with role S is used as U_1 and a user with role T is used as U_2. In both cases, the vulnerable URL is found during crawling with U_1, is not filtered out by the filtering component, and is, therefore, replayed. With V_1, the content similarity determined by the U_1 content similarity validator is 84 ($\geq$ 80) and all content similarities determined by the U_2 content similarity validator are 82 or lower ($<$ 95). With V_2, the similarity values are 100 and 70, respectively. Consequently, V_1 and V_2 are correctly detected.

In particular, V_1 demonstrates once more that it is important to remove navigation areas when determining the similarity value and to also classify non-perfect matches as a similarity match. To illustrate this, Fig. 12 shows the content when accessing the vulnerable URL with users with roles S and T.

First of all, one can see that the navigation areas on the left, on the right and at the top are quite different. Not removing them would result in a similarity value significantly below 80, and V_1 would not be detected. But also the middle parts of the web pages, which are the parts that are actually compared after removing the navigation areas, show some differences, as only the web page on the left side includes the fields Hourly rate and Internal rate. So obviously, although there is indeed a vulnerability, the user with

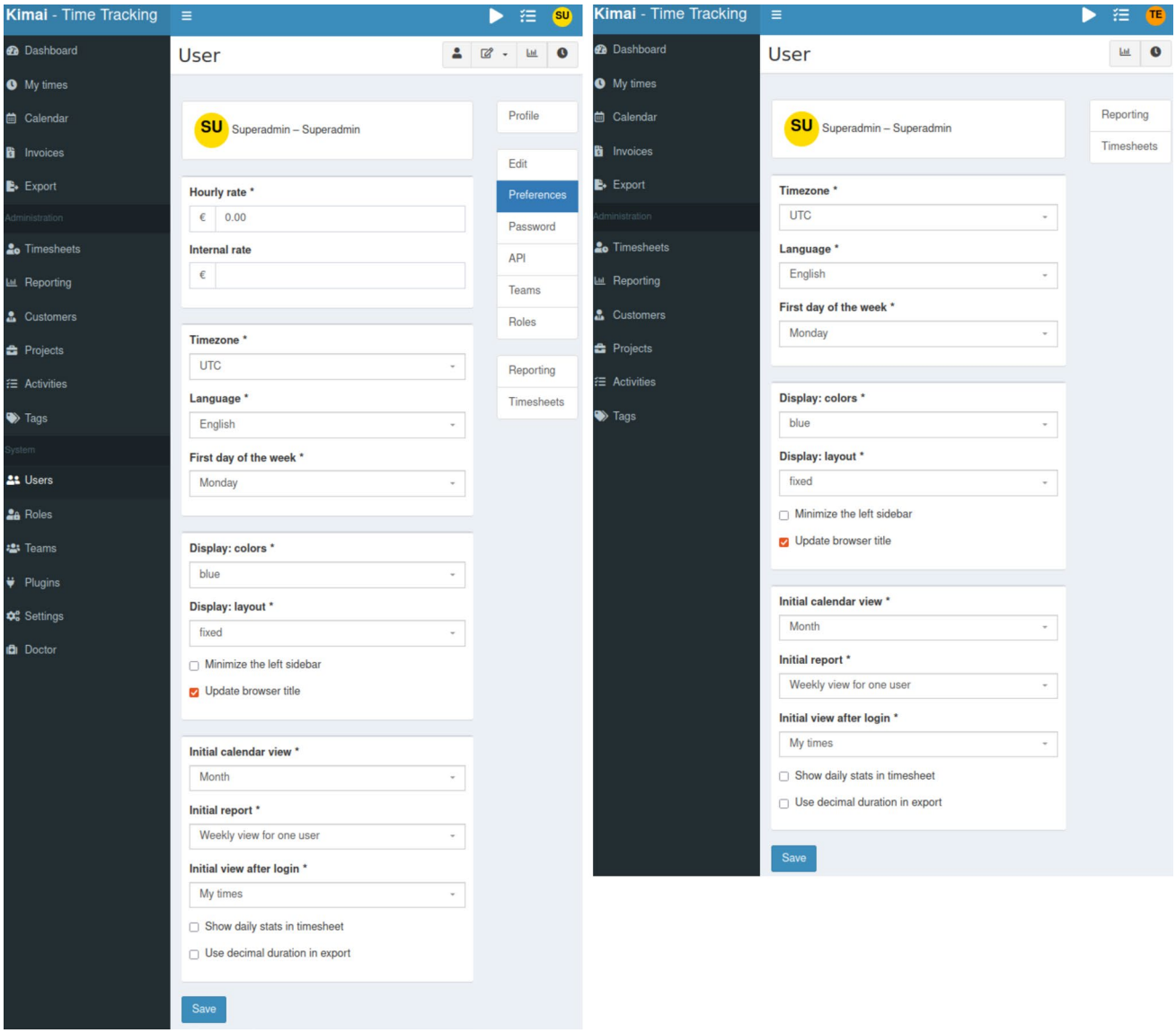

Fig. 12 Responses by Kimai when accessing http://www.site.com/en/profile/superadmin/prefs with role Superadmin (S) (left) and Teamlead (T) (right)

role T does not get access to the complete content of the user with role S and as a result of this, the determined similarity value is "only" 84. If the threshold for similarity matching were set to 100 or very close to 100, no similarity match would occur and again, V_1 would not be detected. But with a default threshold of 80 as set in the configuration file (see Fig. 10), the vulnerability is correctly detected.

One false positive, identified as FP_1, is reported and affects the URL http://www.site.com/en/export/?customers%5B%5D=1 &projects%5B%5D=1 &daterange= &preview=1. This URL is used by U_1 to access a web page that allows to select data to export. U_2 crawls a very similar URL with the same path but with different query parameters, and as a result of this, the URL is not filtered out. Replaying works, the similarity determined by the U_1 content similarity validator is 92 ($\geq$ 80), and the content similarities determined by the U_2 content similarity validator are all 92 or lower ($<$ 95). Consequently, a vulnerability is reported. As U_2 can access basically the same content when navigating the application (using the very similar URL mentioned above), this is indeed a false positive. The highest match of 92 by the U_2 content similarity validator occurs due to a comparison with exactly this very similar URL. Of course, this false positive could easily be prevented by setting the threshold of the U_2 content similarity validator to, e.g., 90 instead of 95. However, 95 has demonstrated to be a good compromise overall so far, and reducing it could introduce other side effects such as missed true vulnerabilities.

Table 7 Summary of evaluation results of Kimai

Vuln.	URL	Detected
V_1	http://www.site.com/en/profile/superadmin/prefs	Yes
V_2	http://www.site.com/en/profile/superadmin/password	Yes
False positives	URL	
FP_1	http://www.site.com/en/export/?customers%5B%5D=1 &projects%5B%5D=1 &daterange= &preview=1	

This seventh and final evaluation shows that sometimes, more configuration effort must be invested to make sure the solution delivers good results as in this case, the *do_not_call_pages* and *tags_to_remove* must be specifically set. Also, together with all other evaluations, this evaluation shows that the content similarity thresholds of 80 and 95 are reasonable compromises that work well in most cases, but sometimes they are set too low or too high for a specific application, which in this case resulted in FP_1. Table 7 summarizes the evaluation results of Kimai.

Discussion and Future Work

The evaluation in "Evaluation" demonstrates that the presented solution is capable of finding access control vulnerabilities in different types of web applications, i.e., in web applications based on different underlying technologies and based on both traditional and modern architectures. Overall, across a selection of seven evaluated applications with a total number of 4'500 detected URLs, 12 of 13 vulnerabilities could be found, only one false negative occurred and in total, five false positives were reported. Compared to the original paper [1], the adaptations made to the overall solution approach resulted in an improved vulnerability detection performance. In particular, the number of false positives could significantly be reduced.

One could argue that the evaluation should also have revealed unknown vulnerabilities in these applications, but none were found. But this is somewhat understandable as all the evaluated applications, with the exception of the very small Marketplace application, are well-established open source applications where significant efforts in security testing have typically already been invested and, consequently, where possibly detected vulnerabilities have already been fixed. Therefore, it is rather unlikely that they still contain standard vulnerabilities such as SQL injection, cross-site scripting or access control vulnerabilities. However, in the context of custom web applications developed by companies for a specific purpose (e.g., for a custom e-shop or to control an industrial system), it is more likely such problems are existing as they are typically less well analyzed with respect to security. Therefore, future evaluations will also take such applications into account.

Although we have shown that the solution approach works to detect access control vulnerabilities, there are still some limitations that provide a lot of potential for future work:

Evaluation scope: With seven applications, the evaluation scope is still limited. While using a relatively small number of applications served well to understand in detail why vulnerabilities could be detected, why false negatives and true positives occurred, and how the overall solution approach could be fine-tuned, an important next step is to test additional web applications, in particular also custom web applications as mentioned above. Also, the two co-authors from scanmeter GmbH will use the solution approach in penetration tests with customers, from which we expect to gain further insights with regard to the true usefulness of the solution in practice.

Fundamental assumptions: If the assumption that web pages presented to a user contain only navigation elements to legitimately accessible content is not valid, then vulnerabilities can be missed (see evaluation of Bludit). Likewise, if the crawler does not find legitimately reachable content or links to such content are missing, there may be false positives (see evaluations of Misago, Magento, Mattermost and Kimai). In general, after evaluating seven web application with a total number of 4'500 URLs detected by the crawling component which resulted in only one false negative and five false positives, it appears that this assumption is mostly correct. However, the general validity of the fundamental assumption can only be demonstrated by analyzing further web applications.

Focus on GET requests: A complete solution should also support POST, PUT, PATCH and DELETE requests. This poses new challenges, in particular as such requests typically change the data and/or state of the web application during crawling and replaying. Resetting the application state regularly can help to a certain degree, but there is still the problem of changes that happen during the actual crawling and replaying. There are further complications such as CSRF tokens, which are often used with requests different from GET. We started to extend the solution so it can also reliably support further request types and the first results are

promising. However, more research is required and we are going to address this in the near future.

General improvements of the entire solution approach: In some areas of the solution, there is still potential to improve the vulnerability detection performance. One such area is the Puppeteer-based crawler. For instance, the false positive FP_2 reported when analyzing Magento can likely be prevented by improving this crawler so it handles dynamically inserted $< a >$ tags correctly. Another area is the implementation of the extraction of textual elements from HTML documents, which works already reasonably well, but which sometimes also identifies web page components as textual elements that are not visible in the browser. These issues will also be addressed in the near future, along with the evaluation of further web applications.

Related Work

Manually driven approaches: A manually driven way to detect access control vulnerabilities in web applications is using interceptor proxies, e.g., *Burp Suite* [22] or *OWASP ZAP* [23]. In such an approach, the security tester manually navigates through the web application using a high-privileged user. Based on this, the proxy learns all URLs and subsequently tries to access all URLs using less privileged users. As a result of this, the security tester is presented with a table that shows which user could access which URLs, which allows the tester to verify the correctness of the implemented access control rules. This approach is supported, e.g., in Burp Suite with the *Autorize* plugin [24] and in OWASP ZAP with the *Access Control Testing* plugin [25]. While all these approaches are black box-based and work well when doing manual security tests, they suffer from their low level of automation.

Automated extraction of the access control model: Another fundamental approach is to analyze a web application with the goal to extract and verify the implemented access control model. In [26], a reverse engineering approach is used to extract a role-based access control model from the source code of a web application, which is then checked to verify whether it corresponds to the access control properties specified by a security engineer. Two limitations with this approach are that the model extraction is dependent on the source code, which means it has to be adapted for every programming language and/or web framework to be supported and that verification of the extracted model is done manually. In [27], the web application under test is analyzed by crawling it using different users. This results in access spaces for the different users and based on this, a machine learning-based approach is employed to derive access rules from an access space. These rules can then be compared automatically with an existing specification, if available. Otherwise, a human expert is involved in the assessment of the access rules. While this approach is independent of the source code of the tested web application, it still requires an access control specification for complete automation.

Replaying-based approaches: Approaches that use some form of replaying have been proposed as well. In [28], the basic idea of crawling a web application with different users and then trying to access all detected URLs with all users to uncover access control vulnerabilities is described. However, the document neither provides details about the entire process nor any evaluation results. In [29], it is stated that the source code of a program often implicitly documents the intended accesses of each role. Based on this, the authors generate sitemaps for different roles from the source code and test, by interacting with the target web application, whether there are roles that allow access to resources that should not be accessible based on the determined sitemaps. In [30], the authors try to extract the access control model from a web application by crawling it with different users. In addition, while crawling, database accesses performed by the web application are monitored. Based on this, the derived access control model describes the relations between users and permitted data accesses, which is then used to create test cases to check whether users can access data that should not be accessible based on this model. In [31], the authors consider typical use cases in a web application (i.e., reasonable sequences of requests instead of random sequences as typically used by crawlers) to improve vulnerability detection accuracy. To do so, an operator first has to manually use the web application to record typical use cases, which are represented in a graph. This graph is then used as a basis to detect vulnerabilities when trying to access resources that should not be accessible by non-privileged users. The approach was applied to one JSP-based web application, where it managed to uncover several vulnerabilities. Finally, in [32], a role-based access control model for an application to be analyzed must be defined manually as a basis. Based on this model, test cases are generated automatically, which can be transformed to executable code to carry out the tests. The approach was applied to three Java programs and demonstrated that it could automatically create a significant portion of the required test code automatically and that it could detect access control defects.

Conclusion

In this paper, we presented a practical solution that allows completely automated black box detection of HTTP GET request-based access control vulnerabilities in web applications. The solution approach has been enhanced compared to the original paper [1], which results in an improved vulnerability detection performance. Also, the evaluation has been

extended from four to seven test applications and the focus of the evaluation has been shifted from a mainly quantitative analysis to providing a better understanding why a specific vulnerability can be detected or why a false negative or false positive occurs.

What separates our solution from previous work is that it is designed to be a truly practical approach that can easily be applied to a wide range of web applications based on traditional or modern architectures, that neither requires access to the source code nor the availability of a formal access control model, and that requires only minimal configuration. The solution was evaluated in the context of seven web applications based on different technologies and managed to detect almost all access control vulnerabilities (all except one) while producing only five false positives. This demonstrates both the effectiveness and the general applicability of the solution approach. There exists a lot of potential for future work, in particular in the context of further analysis and fine-tuning of the solution in the context of various real-world web applications and by extending the solution to support additional request types. We have already started to address these issues.

Acknowledgements This work was partly funded by the Swiss Confederation's innovation promotion agency Innosuissse (projects 31954.1 IP-ICT and 48528.1 IP-ICT).

Funding Open access funding provided by ZHAW Zurich University of Applied Sciences. Partly funded by the Swiss Confederation's innovation promotion agency Innosuissse (projects 31954.1 IP-ICT and 48528.1 IP-ICT).

Availability of data and materials Not applicable.

Declarations

Conflict of interest We declare that we have no conflict of interest.

Code availability Not publicly available.

References

1. Kushnir M, Favre O, Rennhard M, Esposito D, Zahnd V. Automated Black Box Detection of HTTP GET Request-based Access Control Vulnerabilities in Web Applications. In: Proceedings of the 7th International Conference on Information Systems Security and Privacy—ICISSP, Online, 2021; pp 204–216.
2. WhiteHat Security: 2019 application security statistics report 2019. https://info.whitehatsec.com/Content-2019-StatsReport_LP.html. Accessed 18 Sep 2021.
3. Chen S. SECTOOL Market. 2016.http://www.sectoolmarket.com. Accessed 18 Sep 2021.
4. Rennhard M, Esposito D, Ruf L, Wagner A. Improving the effectiveness of web application vulnerability scanning. Int J Adv Internet Technol. 2019;12(1/2):12–27.
5. OWASP. OWASP Top 10 2021. 2021. https://owasp.org/Top10/. Accessed 18 Sep 2021.
6. OWASP. OWASP Top 10—A01:2021—Broken Access Control. 2021. https://owasp.org/Top10/A01_2021-Broken_Access_Control/. Accessed 18 Sep 2021 .
7. Scrapy: Scrapy. 2021. https://scrapy.org. Accessed 18 Sep 2021.
8. Puppeteer: Puppeteer. 2021. https://pptr.dev. Accessed 18 Sep 2021.
9. Kornblum J. Identifying almost identical files using context triggered piecewise hashing. Digit Investig. 2006;3:91–7.
10. Oliver J, Cheng C, Chen Y. TLSH—a Locality Sensitive Hash. In: 2013 Fourth Cybercrime and Trustworthy Computing Workshop, 2013; pp 7–13.
11. Oliver J, Forman S, Cheng C. Using randomization to attack similarity digests. In: International Conference on Applications and Techniques in Information Security, Melbourne, Australia, 2014; pp 199–210.
12. Kornblum J. Source Code of ssdeep.h. 2013. https://github.com/ssdeep-project/ssdeep/blob/master/ssdeep.h. Accessed 18 Sep 2021.
13. mitmproxy: mitmproxy. 2021. https://mitmproxy.org. Accessed 18 Sep 2021.
14. SQLite: SQLite. 2021. https://www.sqlite.org. Accessed 18 Sep 2021.
15. Bludit: Bludit. 2020. https://www.bludit.com. Accessed 18 Sep 2021.
16. Pitoń, R.: Misago. 2020. https://misago-project.org. Accessed 18 Sep 2021.
17. Townsend T. Wordpress Plugin Job Manager. 2015. https://wordpress.org/plugins/job-manager. Accessed 18 Sep 2021.
18. NIST: National Vulnerability Database, CVE-2015-6668. 2015. https://nvd.nist.gov/vuln/detail/CVE-2015-6668. Accessed 18 Sep 2021.
19. Magento Commerce Inc.: Magento. 2021. https://magento.com/products/magento-commerce. Accessed 18 Sep 2021.
20. Mattermost Inc.: Mattermost. 2021. https://mattermost.com. Accessed 18 Sep 2021.
21. Papst K. Kimai. 2021. https://www.kimai.org. Accessed 18 Sep 2021.
22. PortSwigger: Burp Suite. 2021. https://portswigger.net/burp. Accessed 18 Sep 2021.
23. OWASP: OWASP Zed Attack Proxy. 2020. https://owasp.org/www-project-zap. Accessed 18 Sep 2021.
24. Tawly B. Autorize. 2020. https://github.com/portswigger/autorize. Accessed 18 Sep 2021.
25. OWASP: Access Control Testing. 2020. https://www.zaproxy.org/docs/desktop/addons/access-control-testing/. Accessed 18 Sep 2021.
26. Alalfi MH, Cordy JR, Dean TR. Automated verification of role-based access control security models recovered from dynamic

web applications. In: 2012 14th IEEE International Symposium on Web Systems Evolution (WSE), Trento, Italy, 2012; pp 1–10.
27. Le HT, Nguyen CD, Briand L, Hourte B. Automated inference of access control policies for web applications. In: Proceedings of the 20th ACM Symposium on Access Control Models and Technologies. SACMAT '15, Vienna, Austria, 2015; pp 27–37.
28. Segal O. Automated testing of privilege escalation in web applications. 2006. http://index-of.es/Security/testing-privilege-escalation.pdf. Accessed 18 Sep 2021.
29. Sun F, Xu L, Su Z. Static detection of access control vulnerabilities in web applications. In: Proceedings of the 20th USENIX Conference on Security. SEC'11, 2011;11. USENIX Association, USA.
30. Li X, Si X, Xue Y. Automated black-box detection of access control vulnerabilities in web applications. In: Proceedings of the 4th ACM Conference on Data and Application Security And Privacy. CODASPY '14, San Antonio, USA. 2014; pp 49–60.
31. Noseevich G, Petukhov A. Detecting insufficient access control in web applications. In: 2011 First SysSec Workshop, Amsterdam, Netherlands, 2011; pp 11–18.
32. Xu D, Kent M, Thomas L, Mouelhi T, Le Traon Y. Automated model-based testing of role-based access control using predicate/transition nets. IEEE Trans Comput. 2015;64(9):2490–505.