Automated Black Box Detection of HTTP GET Request-based Access Control Vulnerabilities in Web Applications

Malte Kushnir[1], Olivier Favre[1], Marc Rennhard[1], Damiano Esposito[2] and Valentin Zahnd[2]
[1]*Institute of Applied Information Technology, Zurich University of Applied Sciences, Winterthur, Switzerland*
[2]*scanmeter GmbH, Zurich, Switzerland*

Keywords: Automated Web Application Security Testing, Access Control Security Testing, Black Box Security Testing.

Abstract: Automated and reproducible security testing of web applications is getting more and more important, driven by short software development cycles and constraints with respect to time and budget. Some types of vulnerabilities can already be detected reasonably well by automated security scanners, e.g., SQL injection or cross-site scripting vulnerabilities. However, other types of vulnerabilities are much harder to uncover in an automated way. This includes 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 access control vulnerabilities in the context of HTTP GET requests 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 that enables frequent and reproducible testing. An evaluation using four 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 relatively low.

1 INTRODUCTION

Web applications are prevalent in today's world and are used for a wide range of services. Often, web applications provide access to security-critical data or functions and correspondingly, they are frequently attacked (WhiteHat Security, 2019). To counter such attacks, web applications should be developed and operated in a secure way and one important part of this is security testing. 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.

One way to automate security testing of web applications is by using vulnerability scanning. Web application vulnerability scanners test a running application "from the outside" by sending specifically crafted requests to the target and analyzing the received response. Various commercial and open source vulnerability scanners are available (Chen, 2016) and while these tools are certainly not perfect and different tools have different strengths and weaknesses (Rennhard et al., 2019), they are in general well-suited to detect some types of web application vulnerabilities such as SQL injection, cross-site scripting or cross-site request forgery (CSRF) vulnerabilities. However, there are also vulnerability types that are hard to detect for vulnerability scanners, which includes access control vulnerabilities. The main reason for this is that vulnerability scanners typically do not know which users are allowed to do what in a web application, so there's no way to determine which accesses to the web application are legitimate or not.

At the same time, access control vulnerabilities are highly relevant and critical in practice (OWASP, 2017a) and consequently, there's a strong desire for solutions that can at least partly automate detection. Several approaches have been proposed in the past (see Section 6) and while all these proposals have their merits, they usually have limitations that prevent their wide applicability in practice, e.g., because they still require significant manual work, because they require a formal specification of an access control model as a basis (which is typically not available in practice), because they are dependent on the used web application technology (which means they have to be adapted if different technologies are used), or because they require access to the source code (which

Kushnir, M., Favre, O., Rennhard, M., Esposito, D. and Zahnd, V.
Automated Black Box Detection of HTTP GET Request-based Access Control Vulnerabilities in Web Applications.
DOI: 10.5220/0010300102040216
In *Proceedings of the 7th International Conference on Information Systems Security and Privacy (ICISSP 2021)*, pages 204-216
ISBN: 978-989-758-491-6

is not always granted when a web application of a third party is analyzed).

In this paper, we present a practical solution that overcomes the limitations of previous proposals. Our solution is based on existing ideas, which are extended with novel approaches to enable completely automated, black box 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.

The remainder of this paper is organized as follows: Section 2 provides an introduction to access control vulnerabilities in web applications. Section 3 describes our solution to detect such vulnerabilities in an automated way and Section 4 contains the evaluation results. Section 5 discusses the solution approach and the evaluation results and provides directions for future work. Related work is covered in Section 6 and Section 7 concludes this work.

2 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 when 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., by 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 APIs), and vulnerabilities in web application server software and web application frameworks (OWASP, 2017b). The impact of an access control vulnerability depends on the vulnerable component and can range from information disclosure (e.g., an e-banking customer that manages to 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).

3 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- and object-level access control vulnerabilities in the context of HTTP GET requests. The reason for focusing on GET requests is to minimize side effects due to changes in the application state that usually happen with POST, PUT, PATCH or DELETE requests. These requests often change the data and/or state of an application, which can alter its appearance and functionality and which in turn would make it more difficult to compare application behavior when using different users.

The fundamental idea 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 or 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 don't 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. In the next step, it is then checked – for each user – whether content that was found during crawling with the other users (but not with the current user) can be accessed by the current user. If this is the case, then this is a strong indication for an access control vulnerability.

This basic approach is not novel and there exist publications on it or that use of parts of it (see Section 6). 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 by using a strict black approach and by integrating several novel approaches in our overall solution, in particular by combining crawlers with different strengths (see Section 3.3), by using a sequence of several filtering steps to remove HTTP request/response pairs that are not relevant when doing access control tests (see Section 3.4), and by using a sophisticated validation approach to determine whether a possibly detected access control vulnerability is indeed a vulnerability or not (see Section 3.5).

3.1 Prerequisites

The solution is based on two main prerequisites. The first is that one has to know the authentication mechanism used by the web application under test, which can easily be determined by observing the HTTP traffic exchanged between browser and web application. Most web applications use either cookie-based session authentication or token-based authentication with JSON Web Tokens (JWT) and our solution supports both. The second prerequisite is that for each user account which should be included in the analysis, authenticated session IDs or token values can be created and captured. This can be done using the standard sign-up and login functionality of a web application.

3.2 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 are required. These users can have different roles (e.g., a seller and an administrator 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). The workflow described in this and the following subsections is always based on two authenticated users, but it 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 Section 3.7.

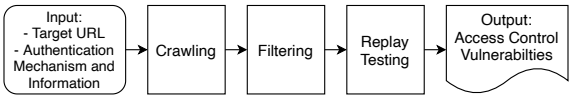

Figure 1: Access control testing workflow.

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 that are not relevant when doing access control tests in the context of the two users. The filtered dataset is then used as input for the ***replay testing component***, which replays requests that were captured with one user with the other user and analyses the behavior of the application to detect vulnerabilities. The result of the replay testing component is a set of detected access control vulnerabilities. The individual components of this workflow are explained in more detail in the following subsections.

3.3 Crawling

The crawling component receives the target URL, the authentication mechanism, and authentication information of two users (identified as U_1 and U_2) as inputs and uses two custom crawlers, built with the popular crawling frameworks ***Scrapy*** (Scrapy, 2020) and ***Puppeteer*** (Puppeteer, 2020). The workflow of this component is shown in Figure 2.

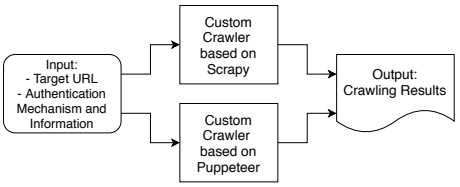

Figure 2: Crawling component.

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 in the browser. Many single page application dynamically load and modify the Document Object Model (DOM) of a web page and can therefore not be reliably crawled with a crawler such as Scrapy that relies on parsing HTML content. 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.

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. Duplicates created during crawling are removed from the lists, i.e., if a URL was 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.

3.4 Filtering

The filtering component uses a sequence of five 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 content that fails the respective check. Note that effective filtering is highly important as otherwise, many false positives will be generated during replay testing. Figure 3 depicts the steps performed during filtering.

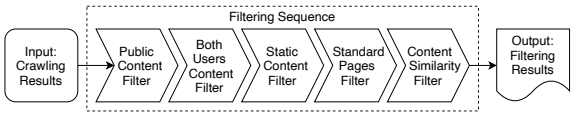

Figure 3: Filtering component.

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 and U_2 with the URLs in the crawling results when no authentication information was used. If there's an overlap, the content is considered public information and therefore filtered out from the crawling results of U_1 and U_2.

The second filter, the ***both users content filter***, removes content that is present in the crawling results of both U_1 and U_2. It compares the URLs of the requests in the crawling results of U_1 and U_2 and if there's an overlap, the content is filtered out from the crawling results of both users. 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 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, logo 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, only CSS and JavaScript files are filtered.

The ***standard pages filter*** filters out specific standard pages. Many web applications have 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.

Finally, the ***content similarity filter*** checks whether the crawling dataset contains HTTP responses that are equal or very similar to each other. The used approach is based on fuzzy hashing and will be described in Section 3.6. This filter mainly aims 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 be detected by this filter and the corresponding requests and responses would be removed from the crawling results. The same applies to other standard content: If multiple URLs in the crawling results point to very similar content, this content has a high probability of being not relevant in access control decisions. For instance, a contact page with support phone numbers or email addresses could be shown when accessing the path */contact* but also when starting a support request from another part of the application.

The output of the filtering component are two lists of filtered HTTP request and response pairs, one for U_1 and one for U_2.

3.5 Replay Testing

The replay testing component is the core component of the solution to detect access control vulnerabilities. It takes the filtered list of HTTP request and response pairs and replays each request with the other user, i.e., requests on the filtered list of U_1 are replayed with U_2 and vice versa. The goal is to learn whether URLs that were found during crawling with one user and that were not filtered out during the previous step can successfully be accessed with the other user, as this is a strong indication of a vulnerability. To verify successful access, the response to the replayed request is analyzed using a series of validators, as illustrated in Figure 4.

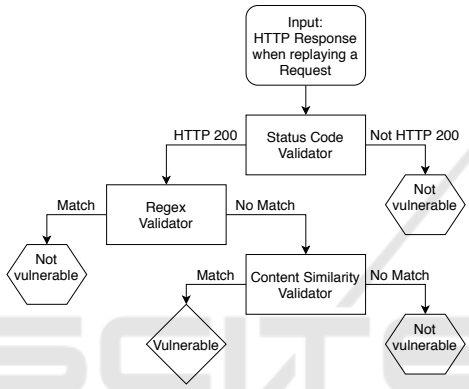

Figure 4: Validators used during replay testing.

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 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 a denied access in a 301, 302, 401 or 403 status code. 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 strings 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's 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 final validator, the ***content similarity validator***, checks whether the content of the response of the replayed request is similar to the response of the original request using the same approach as during the content similarity filtering step. 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's assume that the resource was only found when crawling with U_1. During replay testing, U_2 can successfully access the resource and without this validator, this would wrongly be flagged as a vulnerability. By using this final 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 replay testing, 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 by using the validator, it is recognized that the contents are different, and consequently, no vulnerability is reported.

The output of the replay testing component is a list of requests that correspond to access control vulnerabilities.

3.6 Content Similarity

In Sections 3.4 and 3.5, filters and validators based on content similarity were used to determine the similarity between the content in HTTP responses. Since this is a a key component of our solution, the approach is

described in more detail in this subsection.

There are different approaches to compare the contents of responses for similarity. One option is to directly compare the contents of two responses for equality. While very fast and simple, this has the major drawback that it can't detect even the smallest differences between two responses. In practice, when sending the same request twice, there's 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.

An approach that can cope with small differences is to only compare the relevant parts of the contents for equality. For example, when comparing two HTML documents, one would first remove all elements from the documents that are not interesting for comparison (e.g., meta tags, static content, scripts, footers, etc.) and then compare the stripped documents. The benefit of this solution is that it allows to filter out all elements that could taint a comparison for equality. The big drawback, however, is that one has to define and maintain a list of elements that should be excluded or allowed. In practice, this is difficult and time consuming, especially as there are major differences between different web frameworks.

We therefore use a more general approach that does not require application-specific configuration. 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 are used, *ssdeep* (Kornblum, 2006) and *tlsh* (Oliver et al., 2013). These algorithms were chosen because tlsh has been proven to perform well when comparing HTML documents (Oliver et al., 2014) while ssdeep is one of the best known and most widely used fuzzy hashing algorithms.

The two fuzzy hashing algorithms are used as follows when comparing two contents: First, the ssdeep and tlsh hashes are computed of both contents and compared with each other. If the resulting comparison score is above a certain configurable threshold for either ssdeep or tlsh, the contents are processed by a filtering step where certain elements that are not relevant for the comparison are removed. Then, the fuzzy hashes are computed again, this time of the filtered contents. If the comparison scores for both ssdeep and tlsh are above the threshold, the contents are considered similar. Otherwise, they are considered different.

To illustrate this procedure, we again use the example scenario introduced in Section 3.5 where U_1 and U_2 are two sellers in an e-shop that both can access a resource that lists their own products. The contents of the HTTP responses they get are different and now it must be determined how similar these two contents are to get a final vulnerability verdict. In step one, the fuzzy hashes of both contents are computed and compared with each other. Let's assume that the two contents differ mainly in terms of the textual information about the listed products, while most of the other page elements are the same in both cases due to standard web page components such as scripts, navigation, header, footer, etc. Therefore, it is quite likely that the comparison score is above the threshold for at least one of the fuzzy hash algorithms, which would wrongly indicate a vulnerability. To prevent such false positives, the second step is used, where as many standard components as possible are removed from the contents. In the case of HTML content, a small list of predefined tags is used to define what elements should be removed, which currently contains scripts and meta tags. This list is based on an analysis of our test applications (see Section 4) and is subject to change once more data is available. As a result of this removal, the actual differences (the included products) in the two contents are now more apparent, which results in fuzzy hashes that are much less similar than before and which therefore most likely will result in the correct verdict of *not vulnerable*.

3.7 Testing of Multiple Users and Roles

The workflow described above can be used to detect access control vulnerabilities based on two users. 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 one of the users can also be the anonymous (unauthenticated) user. 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 six runs of the entire workflow would be done based on the pairs *(A,V)*, *(A,S)*, *(A,Y)*, *(V,S)*, *(V,Y)* and *(S,Y)*. Note that if one of the two users is the anonymous user, the entire workflow basically works as described above, but crawling is only done with one authenticated user and the *both users content filter* is omitted.

3.8 Configuration Example

To give an idea about the configuration that is required for the solution to detect vulnerabilities based on two users, Figure 5 shows the default configuration file.

The first two sections *target* and *auth* are self-explanatory and must be specifically configured. As

```
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
  standard_pages: about.php,credits.php
  regex_to_match:
  ssdeep_threshold: 80
  tlsh_threshold: 50
```

Figure 5: Configuration example.

can be seen, the required configuration effort is small. The third section *options* can be adapted to supply additional information about 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 crawler should not follow, e.g., to avoid logging itself 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).
- *ssdeep_threshold/tlsh_threshold*: Thresholds for content similarity matching above which contents are considered similar. The default values *80* and *50* worked well during the evaluation.

3.9 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* (Mitmproxy, 2020). 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* (SQLite, 2020) 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.

4 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 three 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 four web applications that represent a diverse set of traditional and modern web technologies and frameworks:

- *Marketplace*: based on Node.js Express
- *Bludit*: based on PHP
- *Misago*: single page application based on Django (backend) and React (frontend)
- *Wordpress*: based on PHP

4.1 Test Application 1: Marketplace

The first application used for evaluation is a web shop application that is used at our university for educational purposes. The application is quite small in scope and therefore a good starting point to evaluate the approach. The application has an administrative area where users can add and delete products and view and delete the purchases that were made in the shop. There are three different roles that have different permissions in the administrative area:

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

One access control vulnerability was added to the application. It can be found 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 *M* or *S*) can access this resource to add a product.

To evaluate how well our solution performs, all pairs of two roles are considered, including the anonymous user *Y*. This results in six runs of the workflow using pairs *(P,S)*, *(P,M)*, *(P,Y)*, *(S,M)*, *(S,Y)* and *(M,Y)*. In addition, three runs are done using two different users with the same role, i.e., (P_1,P_2), (S_1,S_2) and (M_1,M_2). This corresponds to a full analysis that takes into account all possible combinations. The standard configuration was used for all runs (see Figure 5). Figure 6 shows the results of the crawling and filtering components.

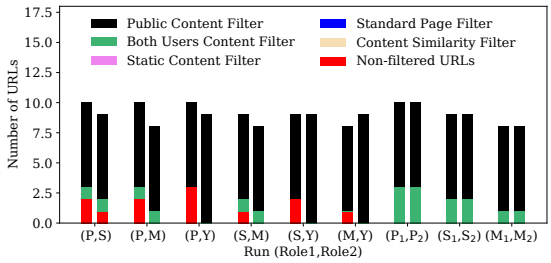

Figure 6: Results of crawling and filtering for Marketplace.

Figure 6 shows two bars for each run because during each run, crawling and filtering is done with both involved users. For instance, looking at the two bars associated with run *(P,S)*, the left bar shows the crawling and filtering results for *P* and the right bar the results for *S*. In general, Figure 6 confirms that the application is indeed small as no more than ten URLs are found in any case. One can also see the effectiveness of the filtering steps that remove most URLs. For instance, looking at the left bar of run *(P,S)*, seven of ten URLs that were found when crawling with *P* are removed by the *public content filter*, which is to be expected as this filter is executed first in the filter chain and as several of the URLs can be reached by the anonymous user. In addition, one URL is removed by the *both users content filter*, as this URL was also found when crawling using the user with role *S*. The other filters have no effect, mainly because the application is so small that all URLs that should be removed could already be eliminated by the first two filters. In the end, only two non-filtered URLs are remaining, one of which corresponds to the URL to access the *Add Product* functionality (where the vulnerability is located). Similar results can be observed in the other runs. Note that in several cases, no non-filtered URLs are left in the end, which corresponds to the correct behavior. For instance both bars of run *(S_1,S_2)* have no non-filtered URLs in the end as with both involved users, the same crawling results are produced and consequently, all URLs are filtered out.

The results of the replay testing component are displayed in Table 1.

Table 1: Results of replay testing for Marketplace.

Run	Vuln exists	Vuln found	False Pos
(P,S)	yes	yes	0
	no	no	0
(P,M)	yes	yes	0
	no	no	0
other	no	no	0
overall	2	2	0

Table 1 shows two rows for each run. For instance, the top row of run *(P,S)* shows the replay testing results when replaying the non-filtered requests of *P* with *S* and the bottom row shows the same when replaying the non-filtered requests of *S* with *P*. The column *Vuln(erability) exists* indicates whether the vulnerability exists in the corresponding scenario, i.e., the entry *yes* in the top row of run *(P,S)* means that the vulnerability allows *S* to access a resource of *P*. Conversely, the entry in the bottom row of run *(P,S)* is set to *no* as there is no vulnerability that allows *P* to access a resource of *S*. The top row of run *(P,S)* also shows that when taking the two URLs that were remaining in the filtered list of URLs of *P* and replaying the corresponding requests with *S*, the vulnerability is detected (column *Vuln(erability) found* is set to *yes*), i.e., access to the *Add Product* resource is indeed possible with *S*. Exactly the same can be observed from the results of run *(P,M)*, i.e., it is also detected that *M* gets access to the *Add Product* resource. The bottom rows of runs *(P,S)* and *(P,M)* show that no vulnerability was detected when accessing the non-filtered URLs of *S* and *M* with *P*, which corresponds to the correct result as no vulnerabilities exist in these two cases. In addition, the rightmost row shows that no false positives were reported during runs *(P,S)* and *(P,M)*. In the other seven runs, no vulnerabilities exist and none are reported (also no false positives). Therefore, the results of these runs are summarized in a single row with the *Run* column set to *other*.

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 in both cases where it was present without reporting any false positives.

4.2 Test Application 2: Bludit

Bludit (Bludit, 2020) is a CMS that represents a more traditional web application, where most of the logic is implemented in the backend. It has an extensive privileged area where authenticated user can add, modify and delete content. There are three roles:

- Role *Author (U)*: create, modify and delete own content
- Role *Editor (E)*: create, modify and delete any content of any user
- Role *Administrator (A)*: all permissions of *E*, in addition 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, every authenticated user is able to access the resource where all plugin settings can be viewed and modified.

- V_2 is similar to V_1 but exists in a different resource, namely the page where a user with role *A* can view and change general website settings.
- V_3 allows authenticated users with role *U* to get access to resources where content of other users with role *U* can be modified.

Nine runs were done that consider all possible pairs based on the three roles and the anonymous user *Y*. To prevent state changes in the application during testing (despite using only GET requests), some resources were added to the *do_not_call_pages* configuration (specifically: *install-plugin*, *install-theme*, *uninstall-plugin*, *uninstall-theme*). Otherwise, the standard configuration was used. Figure 7 shows the results of the crawling and filtering components.

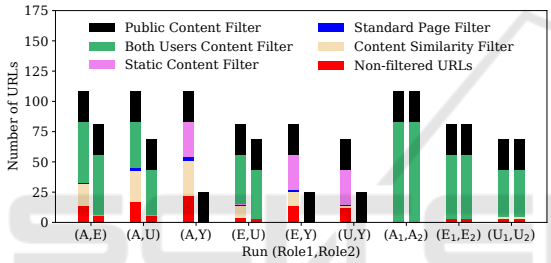

Figure 7: Results of crawling and filtering for Bludit.

With a maximum of over 100 unique URLs identified during crawling, this application is much bigger than the first one and therefore better suited to evaluate the effectiveness of the filtering steps. Compared to test application 1, the publicly reachable resources make up a much smaller part of the overall application. Only about 20% is accessible by an anonymous user and filtered out by the *public content filter*. The most significant filtering happens with the *both users content filter* with the exception of the runs that involve the anonymous user *Y*. The reason for the latter is that in theses runs, content that can be reached by both users corresponds to public content, which is already filtered out by the *public content filter*. The *content similarity filter* is also effective, mainly as there are a lot of menu pages in the protected area of the application that are very similar and that often differ only in one HTML tag. Overall, the number of URLs could be reduced by more than 90% on average.

The results of the replay testing component are displayed in Table 2. The results show that V_1 and V_3 were detected whenever they should have been detected in the corresponding runs. However, V_2 was never detected because the vulnerable URL was consistently filtered out by the *both users content filter*. The reason for this is a strange application design choice by the developers: 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 isn't permitted to visit. Therefore, during crawling, the vulnerable URL can be found by all authenticated users and not only by *A* and consequently, it is filtered out.

Table 2: Results of replay testing for Bludit.

Run	Vuln exists			Vuln found			False Pos
	V_1	V_2	V_3	V_1	V_2	V_3	(new unique)
(A,E)	yes	yes	no	yes	no	no	0
	no	no	no	no	no	no	0
(A,U)	yes	yes	yes	yes	no	yes	0
	no	no	no	no	no	no	0
(E,U)	no	no	yes	no	no	yes	2 (2)
	no	no	no	no	no	no	2 (0)
(E_1,E_2)	no	no	no	no	no	no	2 (0)
	no	no	no	no	no	no	2 (0)
(U_1,U_2)	no	no	yes	no	no	yes	2 (0)
	no	no	yes	no	no	yes	2 (0)
other	no	no	no	no	no	no	0
overall	2	2	3	2	0	3	2 unique

Overall, 12 false positives were reported, However, they correspond to only two unique false positives. This is reflected in the rightmost column in Table 2: Only the first two false positives are actually counted as they are new unique false positives (the new unique false positives are shown in parentheses). All other reported false positives are duplicates of the first two and not counted as unique false positives. The total number of unique false positives is also listed in the bottom row. The two false positives are application-specific and related to editing the profile of a user: When accessing the user-specific URL to edit the profile of another user, the web application returns the page to edit the own profile instead, so the own profile is accessible with the URL belonging to the profile of another user. A more standard behavior would be a redirect to the correct URL of the current user or a status code such as 403 or 404.

4.3 Test Application 3: Misago

Misago (Pitoń, 2020) provides a web forum solution. 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 crawler based on Puppeteer 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 approach can handle non-HTML content, as communication between front- and backend is mainly done using JSON data. 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 actions 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 any authenticated user to view the complete list of users of the forum.
- 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 client-side navigation URL and in the REST API in the backend, which serves the content of the thread.
- 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 client-side 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.

Nine runs were done using all pairs based on the three roles and the anonymous user *Y*. The standard configuration was used for all runs. Figure 8 shows the results of the crawling and filtering components.

Figure 8 shows that the number of URLs has increased again compared to the previous applications. Also, the number of URLs remaining after filtering is significantly higher than before, both in absolute and relative terms. This is not because of a flawed filtering process, but is a consequence of the fact that the permissions between the roles differ significantly. In particular, *A* can access many resources that no other user can access, which can clearly be seen in Figure 8.

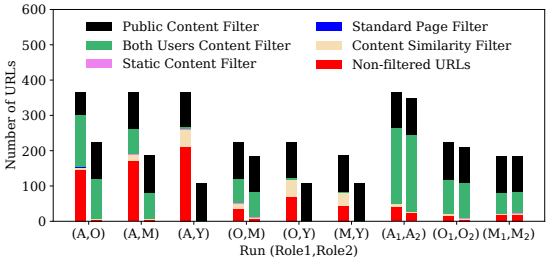

Figure 8: Results of crawling and filtering for Misago.

The results of the replay testing component are displayed in Table 3.

Table 3: Results of replay testing for Misago.

Run	Vuln exists V_1	V_2	V_3	Vuln found V_1	V_2	V_3	False Pos (new unique)
(A,O)	yes	yes	no	yes	yes	no	7 (7)
	no	no	no	no	no	no	2 (1)
(A,M)	yes	yes	no	yes	yes	no	2 (0)
	no	no	no	no	no	no	2 (0)
(A,Y)	no	no	no	no	no	no	5 (1)
	no	no	no	no	no	no	0
(O,M)	no	yes	yes	no	yes	yes	0
	no	yes	yes	no	yes	yes	1 (0)
(O,Y)	no	no	no	no	no	no	1 (0)
	no	no	no	no	no	no	0
(M,Y)	no	no	no	no	no	no	1 (0)
	no	no	no	no	no	no	0
(A_1,A_2)	no	yes	yes	no	yes	yes	14 (7)
	no	no	yes	no	no	yes	7 (1)
(O_1,O_2)	no	yes	yes	no	yes	yes	1 (0)
	no	no	yes	no	no	yes	2 (1)
(M_1,M_2)	no	yes	yes	no	yes	yes	1 (1)
	no	no	yes	no	no	yes	6 (1)
overall	2	7	8	2	7	8	20 unique

All vulnerabilities were detected whenever they should have been detected in the corresponding run. Overall, 20 unique false positives were reported, but considering the large number of URLs that were tested (all non-filtered URLs shown in Figure 8), the number of false positives is relatively small. Several of the false positives occurred because some user profiles that are accessible by less privileged users were not found when crawling using these less privileged users, as the corresponding users had neither created a thread nor a forum post and consequently, no links to the profiles were found. This shows a design weakness in our approach as it assumes that ideally, the web application under test is showing links to all content a specific user has access to. Most of the other false positives can be attributed to crawling issues, where resources were not found either due to application errors or application-specific behavior. For ex-

ample, the password reset page is only found when crawling with an authenticated user, but during replay testing, it can also be accessed with the anonymous user, although an anonymous user has no need for a password reset. Some of these issues could be mitigated by adding the corresponding resources under *do_not_call_pages* in the configuration file.

4.4 Test Application 4: Wordpress

The final test application is a standard Wordpress site with a vulnerable version of the plugin *Job Manager* (Townsend, 2015). The plugin allows to list job offers and lets users of the site submit job offers, and contains a published access control vulnerability (CVE-2015-6668 (NIST, 2015)). The vulnerability 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 are even accessible by anonymous users.

In the evaluation, the role *Author (U)* and the anonymous user *Y* are considered. An author has access to his own job applications but not to the applications of others. Two users with the role *U* are included in the analysis, where U_1 has created one job application with an attached PDF document and U_2 hasn't created any job applications. Based on this, three runs were done. The standard configuration was used for all runs. Figure 9 shows the results of the crawling and filtering components.

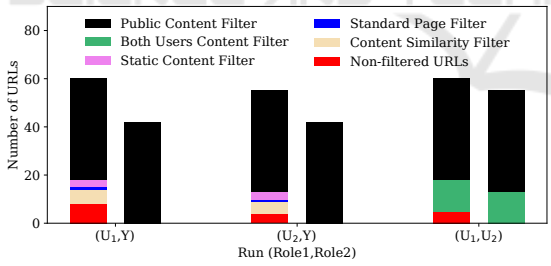

Figure 9: Results of crawling and filtering for Wordpress.

The crawling component identified at most 60 URLs and the different filtering steps managed to reduce the number of URLs significantly.

The results of the replay testing component are displayed in Table 4. The results show that the vulnerability could be detected in both cases where it should have been detected, i.e., both *Y* and U_2 managed to get access to the PDF file in the job application offered by U_1. No false positives were reported.

Table 4: Results of replay testing for Wordpress.

Run	Vuln exists	Vuln found	False Pos
(U_1,Y)	yes	yes	0
	no	no	0
(U_2,Y)	no	no	0
	no	no	0
(U_1,U_2)	yes	yes	0
	no	no	0
overall	2	2	0

5 DISCUSSION AND FUTURE WORK

As demonstrated in Section 4, 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, only one vulnerability was missed and the number of false positives is relatively low. However, there are also some limitations that provide a lot of potential for future work.

Evaluation Scope. The current evaluation scope is limited. To get a better understanding about the general applicability of the solution, we will apply it to further web applications. In particular, the two co-authors from scanmeter GmbH will use the prototype implementation in penetration tests with customers, from which we expect to gain lots of insights with regard to the 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 overlooked (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 evaluation of *Misago*). To get a better understanding of how prevalent such issues are in real-world web application and how to deal with these issues, more analysis is required and we expect that the broader application of the prototype implementation by scanmeter GmbH will provide further insights with respect to these issues as well. Based on the findings, we will then extend and/or fine-tune the overall solution approach to optimize its results. One possible approach in this regard is to use fuzzing techniques to find content that is not directly linked in the web pages presented to the user. Such content is often accessed via request parameters, e.g., https://www.site.com/invoices?id=1. By analyzing the crawled URLs for request parameters and using fuzzing techniques

on these parameters it should be possible to extend the coverage of our testing approach.

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 replay testing. 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 replay testing. There are further complications such as CSRF tokens, which are often used with requests different from GET. Extending our solution so it can also reliably support further request types is an open research problem and one that we are going to address in the near future.

6 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* (PortSwigger, 2020) or *OWASP ZAP* (OWASP, 2020b). 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 (Tawly, 2020) and in OWASP ZAP with the *Access Control Testing* plugin (OWASP, 2020a). 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 (Alalfi et al., 2012), 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 (Le et al., 2015), 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.

Replay Testing-based Approaches. Approaches that use some form of replay testing have been proposed as well. In (Segal, 2006), 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 (Sun et al., 2011), 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 shouldn't be accessible based on the determined sitemaps. In (Li et al., 2014), 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 (Noseevich and Petukhov, 2011), 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 (Xu et al., 2015), 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 sig-

nificant portion of the required test code automatically and that it could detect access control defects.

7 CONCLUSIONS

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. What separates our solution from previous work is that it is designed to be a truly practical approach in the sense that it is easily applicable to a wide range of web applications based on traditional or modern architectures, that it neither requires access to the source code nor the availability of a formal access control model, and that it requires only minimal configuration. The solution was evaluated in the context of four web applications based on different technologies and managed to detect almost all access control vulnerabilities (all except one) while producing only relatively few 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 are going to address these issues in the near future.

ACKNOWLEDGEMENTS

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

REFERENCES

Alalfi, M. H., Cordy, J. R., and Dean, T. R. (2012). 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)*, pages 1–10, Trento, Italy.

Bludit (2020). Bludit. https://www.bludit.com.

Chen, S. (2016). SECTOOL Market. http://www.sectoolmarket.com.

Kornblum, J. (2006). Identifying Almost Identical Files Using Context Triggered Piecewise Hashing. *Digital Investigation*, 3:91 – 97.

Le, H. T., Nguyen, C. D., Briand, L., and Hourte, B. (2015). Automated Inference of Access Control Policies for Web Applications. In *Proceedings of the 20th ACM Symposium on Access Control Models and Technologies*, SACMAT '15, pages 27–37, Vienna, Austria.

Li, X., Si, X., and Xue, Y. (2014). 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, pages 49–60, San Antonio, USA.

Mitmproxy (2020). Mitmproxy. https://mitmproxy.org.

NIST (2015). National Vulnerability Database, CVE-2015-6668. https://nvd.nist.gov/vuln/detail/CVE-2015-6668.

Noseevich, G. and Petukhov, A. (2011). Detecting Insufficient Access Control in Web Applications. In *2011 First SysSec Workshop*, pages 11–18, Amsterdam, Netherlands.

Oliver, J., Cheng, C., and Chen, Y. (2013). TLSH – A Locality Sensitive Hash. In *2013 Fourth Cybercrime and Trustworthy Computing Workshop*, pages 7–13.

Oliver, J., Forman, S., and Cheng, C. (2014). Using Randomization to Attack Similarity Digests. In *International Conference on Applications and Techniques in Information Security*, pages 199–210. Springer.

OWASP (2017a). OWASP Top Ten. https://owasp.org/www-project-top-ten.

OWASP (2017b). OWASP Top Ten - A5:2017 Broken Access Control. https://owasp.org/www-project-top-ten/OWASP_Top_Ten_2017/Top_10-2017_A5-Broken_Access_Control.

OWASP (2020a). Access Control Testing. https://www.zaproxy.org/docs/desktop/addons/access-control-testing/.

OWASP (2020b). OWASP Zed Attack Proxy. https://owasp.org/www-project-zap.

Pitoń, R. (2020). Misago. https://misago-project.org.

PortSwigger (2020). Burp Suite. https://portswigger.net/burp.

Puppeteer (2020). Puppeteer. https://pptr.dev.

Rennhard, M., Esposito, D., Ruf, L., and Wagner, A. (2019). Improving the Effectiveness of Web Application Vulnerability Scanning. *International Journal on Advances in Internet Technology.*, 12(1/2):12–27.

Scrapy (2020). Scrapy. https://scrapy.org.

Segal, O. (2006). Automated Testing of Privilege Escalation in Web Applications. http://index-of.es/Security/testing-privilege-escalation.pdf.

SQLite (2020). SQLite. https://www.sqlite.org.

Sun, F., Xu, L., and Su, Z. (2011). Static Detection of Access Control Vulnerabilities in Web Applications. In *Proceedings of the 20th USENIX Conference on Security*, SEC'11, page 11, USA. USENIX Association.

Tawly, B. (2020). Autorize. https://github.com/portswigger/autorize.

Townsend, T. (2015). Wordpress Plugin Job Manager. https://wordpress.org/plugins/job-manager.

WhiteHat Security (2019). 2019 Application Security Statistics Report. https://info.whitehatsec.com/Content-2019-StatsReport_LP.html.

Xu, D., Kent, M., Thomas, L., Mouelhi, T., and Le Traon, Y. (2015). Automated Model-Based Testing of Role-Based Access Control Using Predicate/Transition Nets. *IEEE Transactions on Computers*, 64(9).