A2CT: Automated Detection of Function and Object-Level Access Control Vulnerabilities in Web Applications

Michael Schlaubitz[a], Onur Veyisoglu and Marc Rennhard[b]
Institute of Computer Science (InIT), ZHAW Zurich University of Applied Sciences, Winterthur, Switzerland

Keywords: Automated Web Application Security Testing, Access Control Vulnerabilities, Zero-Day Vulnerabilities.

Abstract: In view of growing security risks, automated security testing of web applications is getting more and more important. There already exist capable tools to detect common vulnerability types such as SQL injection or cross-site scripting. Access control vulnerabilities, however, are still a vulnerability category that is much harder to detect in an automated fashion, while at the same time representing a highly relevant security problem in practice. In this paper, we present A2CT, a practical approach for the automated detection of access control vulnerabilities in web applications. A2CT supports most web applications and can detect vulnerabilities in the context of all HTTP request types (GET, POST, PUT, PATCH, DELETE). To demonstrate the practical usefulness of A2CT, an evaluation based on 30 publicly available web applications was done. Overall, A2CT managed to uncover 14 previously unknown vulnerabilities in two of these web applications, which resulted in six published CVE records. To encourage further research, the source code of A2CT is made available under an open-source license.

1 INTRODUCTION

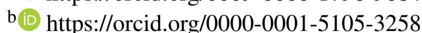
Web applications are attractive attack targets (Verizon, 2023). Consequently, they should be tested with regard to security, with the goal to detect vulnerabilities so they can be remedied. If possible, security testing should be automated, as this allows for more efficient and reproducible security tests.

A prominent automated security testing method is vulnerability scanning. In web applications, vulnerability scanners can detect some vulnerability types by sending HTTP requests to the running web application and analyzing the received responses. There exist several vulnerability scanners (Bennets, 2023; OWASP, 2024) that can detect, e.g., SQL injection and cross-site scripting vulnerabilities. A vulnerability type that is much harder to detect is access control vulnerabilities. The main challenge is that vulnerability scanners usually do not know the access rights of the different users, so there is no basis to determine legitimate or illegitimate accesses during scanning.

Several approaches to automatically detect access control vulnerabilities have been proposed (see Section 6), but they typically have limitations that prevent their adoption in practice. These limitations include, e.g., that significant manual work is required, that a formal specification of an access control model is needed (which is rarely available in practice), or that the approach relies on a specific web application technology (which limits its broad applicability).

This lack of a truly practical automated testing methodology is in stark contrast to the relevance of access control vulnerabilities. According to the Open Worldwide Application Security Project (OWASP), 94% of web applications contain some form of access control issue, more than any other vulnerability type (OWASP, 2021a). As a result, the vulnerability type *broken access control* climbed from fifth to first place on OWASP's current top ten list of the most critical vulnerability types in web applications (OWASP, 2021b). This underlines the need for automated detection approaches for this vulnerability type.

In this paper, we present A2CT (*Automated Access Control Tester*), an automated access control vulnerability detection approach for web applications that overcomes the limitations of previous proposals. A2CT supports all relevant HTTP request types (GET, POST, PUT, PATCH, DELETE) and can test web applications based on different technologies and architectural styles. Our evaluation shows that A2CT can detect previously unknown vulnerabilities.

[a] https://orcid.org/0009-0000-1796-9684
[b]  https://orcid.org/0000-0001-5105-3258

Schlaubitz, M., Veyisoglu, O. and Rennhard, M.
A2CT: Automated Detection of Function and Object-Level Access Control Vulnerabilities in Web Applications.
DOI: 10.5220/0013092700003899
Paper published under CC license (CC BY-NC-ND 4.0)
In *Proceedings of the 11th International Conference on Information Systems Security and Privacy (ICISSP 2025) - Volume 2*, pages 425-436
ISBN: 978-989-758-735-1; ISSN: 2184-4356
Proceedings Copyright © 2025 by SCITEPRESS – Science and Technology Publications, Lda.

Specifically, 14 access control vulnerabilities in two web applications were found, which resulted in six newly published CVE records (MITRE, 2024). In summary, our main contributions are the following:

- A2CT, a solution to automatically detect access control vulnerabilities in web applications that requires only minimal configuration and that does not require a formal access control model.
- An evaluation of A2CT using 30 web applications, which revealed 14 access control vulnerabilities in two of these applications. This demonstrates the practical value of A2CT.
- The source code of A2CT, made available to the research community under an open-source license (ZHAW Infosec Research Group, 2024).

This work is based on a previous version developed by our research group (Kushnir et al., 2021; Rennhard et al., 2022). However, the approach and the evaluation have been significantly extended and improved, and the previous limitations have been successfully addressed (see Sections 3 and 4).

The remainder of this paper is organized as follows: In Section 2, an introduction to access control vulnerabilities in web applications is given. Next, Section 3 explains how A2CT works to automatically detect access control vulnerabilities. In Section 4, the evaluation results are presented, followed by a discussion of A2CT in Section 5, Related work is covered in Section 6, and Section 7 concludes this work.

2 DETECTABLE ACCESS CONTROL VULNERABILITIES

According to PortSwigger, *"access control is the application of constraints on who or what is authorized to perform actions or access resources"* (Portswigger, 2024). If the mechanisms to enforce this in a web application contain vulnerabilities, users may be able to act outside of their intended permissions.

The impacts of access control vulnerabilities can be manifold. Examples include information disclosure (e.g., an e-banking customer who can view account details of other customers), unauthorized manipulation of data (e.g., a user of a web shop who can modify the shopping cart of other users), and getting complete control of a web application (e.g., an attacker manages to access the administrator panel).

There are different types of access control vulnerabilities, for an overview, see (Portswigger, 2024). The two most prominent types are *function-level* and *object-level* access control vulnerabilities. A2CT can detect both types, and they are described as follows:

- A function-level access control vulnerability means a user gets illegitimate access to a specific function in a web application. For example, a URL like https://site.com/admin/view-users, which provides the function to view all registered users, could be intended to be accessed only by administrative users. If this resource admin/view-users could also be accessed successfully by non-administrators, this would correspond to a function-level access control vulnerability.
- An object-level access control vulnerability means a user has legitimate access to a function in a web application, but can gain unauthorized access to objects in the context of this function. For instance, a seller in a web shop may have legitimate access to a function to edit his own products via the URL https://site.com/edit-product&id=123 as the product with id 123 is one of his own products. If the seller were to change the id in the URL so that it corresponds to a product of another seller (e.g., https://site.com/edit-product&id=456), and if the seller gains illegitimate access to edit this product, then this would correspond to an object-level access control vulnerability.

3 A2CT: AUTOMATED ACCESS CONTROL TESTER

A2CT uses a black box approach: a running web application is tested by interacting with it from the outside, without any knowledge of internal information such as the source code or the access control model. Figure 1 illustrates a typical setup how A2CT is used, which corresponds to the setup we used during development and for the evaluation (see Section 4).

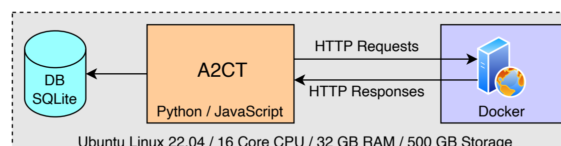


Figure 1: A2CT setup.

A2CT (implemented in Python and JavaScript) and the web application under test run on the same system, and the web application runs in a *Docker* container (Docker, 2024). This setup allows to easily reset the web application during a test (see Section 3.1). A2CT interacts with the web application by sending HTTP requests and receiving HTTP responses. The request/response pairs are stored in an *SQLite* database (SQLite, 2024). The underlying system we used was based on Ubuntu Linux 22.04 with

a 16 core CPU, 32 GB RAM and 500 GB storage.

The fundamental idea how A2CT can determine permitted access relies on the assumption that in many web applications, the web pages presented to a user contain only buttons, links and other navigation elements that correspond to functions that can be legitimately accessed by this user. To illustrate this, assume that the dashboard for administrators contains various user interface (UI) elements to access different administrator functions. Typically, access to the dashboard and the included UI elements is only possible after an administrator has logged in. Conversely, for a standard (non-administrator) user, these UI elements are typically not reachable. However, if a standard user manages to successfully access an administrator function by directly issuing the corresponding HTTP request manually (i.e., without using the UI), then an access control vulnerability is likely the cause. Based on this assumption, web crawling can be used to automatically determine which requests can be legitimately issued and which content can be legitimately accessed by any given user, as crawlers follow links and click buttons to emulate a human user interacting with a web application.

As stated in Section 1, A2CT is based on a previous version developed by our research group that has been significantly extended and improved. The improvements are mainly in the following areas, more details will be provided in the following subsections:

- **Improved Testing Coverage by Supporting all HTTP Request Types:** The previous version supported only HTTP GET requests, i.e., requests that read information from a web application. This was extended to support also request types that modify the state of the web application. Consequently, all relevant HTTP request types are now supported (GET, POST, PUT, DELETE, PATCH) and as a result, it is now possible to detect vulnerabilities in the context of all request types. This extension had an impact on the requirements (see Section 3.1) and all components of A2CT (see Sections 3.3-3.6).
- **Improved Testing Coverage by Replacing and Redesigning the Crawling Component:** The crawling component (see Section 3.3) of A2CT has a big impact on the potential to detect vulnerabilities. Compared to the previous version, this component was redesigned and significantly enhanced. As a result, many more areas of a web application can be reached by the crawling component, making it possible to detect vulnerabilities in areas that could not be tested previously.
- **Improved Authentication Method of the Crawling Component:** Authenticating as a specific user in the web application under test has been moved to the browser instance which also used to do the crawling. This ensures that this browser instance is correctly initialized with the authentication information before crawling begins, which is more robust than the previous version where authentication was done outside this browser instance.
- **Optimized Filtering Component:** The filtering component (see Section 3.4) was extended with a new deduplication filter. This allows to remove duplicate HTTP request/response pairs from the crawling result and also helps to better understand the results of the crawling process and why A2CT has reported a specific vulnerability.
- **More Efficient Overall Testing Process:** The previous version supported only two users at a time and tests with more users required several individual test runs. The current version can include any number of users in a test run, which is much more efficient, as crawling with a particular user must only be done once.

3.1 A2CT Requirements

To use A2CT, two requirements must be fulfilled:

- The authentication mechanism of the web application under test can be automated using an authentication script.
- The web application under test can be reset automatically using a reset script.

The first requirement is because for every user included in a test, A2CT must be able to log into the web application with the user's credentials (e.g., user name and password). Otherwise, accessing the web application using a specific user identity would not be possible. The easiest way to create an authentication script is by doing a manual login using the browser and recording the login steps with a technology such as *Playwright codegen* (Playwright, 2024b). Typically, these login steps include instructions to use a URL to navigate to the login page of the web application and to submit a POST request with the user credentials. The recorded login steps can then be added to an authentication script template, and the resulting script can be used to perform authentication directly within a browser instance (see Section 3.3).

The second is a newly introduced requirement because A2CT now also supports request types (POST, PUT, DELETE, PATCH) that can modify the state of the web application under test, i.e., these requests may create, change or delete data and resources. This reduces the reliability of A2CT because resources that

were available when accessing the web application with one user may be no longer be present (or contain other data) when accessing them later with a different user. To reduce the negative effects of such state changes, the web application is reset to its initial state whenever A2CT changes the user that is used to access the web application. Resetting the web application implies restoring the database to its original state and restarting the web application. This is done by a reset script, which executes the necessary commands. The most robust way to do this is by containerizing the web application with technologies like *Docker* (as we did, see Figure 1).

For example authentication and reset scripts, refer to (ZHAW Infosec Research Group, 2024).

3.2 A2CT Overview and Workflow

Besides the authentication and reset scripts, A2CT also requires a configuration file. Listing 1 shows an example configuration file for a web shop application.

```
1  target:
2    target_url: http://172.17.0.1:8080
3    auth_script: ./auth_scripts/auth.js
4    reset_script: ./reset_scripts/reset.py
5  auth:
6    users:
7      - admin: Lo7Y.oCe?ULa
8      - seller1: xIMp6aSTw-ue
9      - seller2: uGm_natE&r2m
10   combinations:
11     type: selected
12     user_pairs:
13       - admin public
14       - admin seller1
15       - seller1 public
16       - seller1 seller2
17 options:
18   do_not_call_pages: logout|login|signup|password
19   static_content_extensions: js,css,img,jpg,png,svg,gif
20   standard_pages: about.php,credits.php
21   regex_to_match: access denied|unauthorized
```

Listing 1: Configuration file example.

The configuration file uses YAML format. The *target* node contains the base URL of the web application under test and the file paths to the authentication and reset scripts. The *auth* node contains the users that should be included in the test and their passwords (*users* node). Here, three users are included: an administrator of the web shop (*admin*) and two sellers (*seller1* and *seller2*). In addition, user *public* is always included implicitly. The *public* user identifies the non-authenticated user, i.e., the user that accesses the web application without logging in.

The *auth* node also contains a *combinations* node with the user combinations to be tested. By using *type: all*, all combinations of two users would be tested. In this example, *type: selected* is used, which allows to define the user combinations in the *user_pairs* node. Each entry contains two users and specifies that it should be tested if the second user can illegitimately access resources of the first user. For instance, with *admin public*, it is tested if user *public* can illegitimately access resources of user *admin*. Likewise, *admin seller1* tests if user *seller1* can access administrative resources that should not be accessible, and *seller1 public* tests if user *public* can illegitimately access resources of user *seller1*. As these first three entries include user pairs with different privilege levels, they mainly serve to detect function-level access control vulnerabilities. On the other hand, the final entry *seller1 seller2* includes two users of the same privilege level (i.e., they have the same role) to test if user *seller2* can access resources of user *seller1* that should not be accessible. This mainly allows to uncover object-level access control vulnerabilities. The *options* node contains further configurations that will be explained in the following subsections.

Based on the configuration file and the authentication and reset scripts, the overall A2CT workflow consists of four sequential components: *crawling component*, *filtering component*, *replaying component*, and *validation component*, as illustrated in Figure 2.

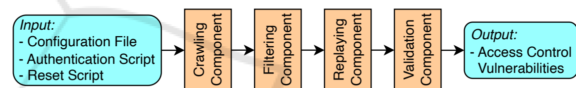


Figure 2: The four components of the A2CT workflow.

In the first step of the workflow, the crawling component crawls the web application with all users specified in the configuration file (and also the *public* user) to record the reachable content of each user. Then, the filtering component receives the data from the crawling component, filters out irrelevant entries, and prepares the input for the next component. After that, the replaying component uses each user pair defined in the *user_pairs* node in the configuration file to test if resources that could be reached by the first but not by the second user during crawling can be directly accessed by the second user. Finally, the validation component assesses the results of the replaying component to determine whether a potential access vulnerability has been detected. The output of the validation component are HTTP request/response pairs where a potential access control vulnerability has been detected. All four components will be explained in more detail in the following subsections.

Algorithm 1 describes the entire workflow in a more formal way. When describing the four components in the following subsections, we will refer to the corresponding lines in this algorithm, and also to the configuration file in Listing 1.

Input: configuration file, authentication script, reset script
Output: list of vulnerable request/response pairs

1. **forall** $U_i \in users$ **do**
2. reset web application
3. $B_{U_i} \leftarrow$ authenticate U_i ▷ authenticated browser instance B_{U_i}
4. $C_{U_i} \leftarrow$ crawl(U_i, B_{U_i}) ▷ crawling results C_{U_i}
5. **end**
6. **forall** $U_i \in users$ **do**
7. $F_{U_i} \leftarrow$ deduplication_filter(C_{U_i})
8. $F_{U_i} \leftarrow$ public_content_filter(F_{U_i})
9. $F_{U_i} \leftarrow$ static_content_filter(F_{U_i})
10. $F_{U_i} \leftarrow$ standard_pages_filter(F_{U_i}) ▷ filtering results F_{U_i}
11. **end**
12. **forall** $(U_1, U_2) \in user_pairs$ **do**
13. $F_{U_1,U_2} \leftarrow$ other_user_content_filter(F_{U_1}, F_{U_2}) ▷ filt. resuls F_{U_1,U_2}
14. **end**
15. **forall** $(U_1, U_2) \in user_pairs$ **do**
16. reset web application
17. $B_{U_2} \leftarrow$ authenticate U_2
18. $R_{U_1,U_2} \leftarrow$ replay_requests(F_{U_1,U_2}, B_{U_2}) ▷ replaying results R_{U_1,U_2}
19. **end**
20. **forall** $(U_1, U_2) \in user_pairs$ **do**
21. **forall** $R_{U_1,U_2}[i] \in R_{U_1,U_2}$ **do**
22. **if not** *status_code_validator*($R_{U_1,U_2}[i]$) **then**
23. **continue** ▷ not vulnerable
24. **end**
25. **if** $R_{U_1,U_2}[i]$ *is HTTP 3xx* **then**
26. **if** *redirect_validator*($R_{U_1,U_2}[i]$) **then**
27. flag vulnerability ▷ vulnerable
28. **continue**
29. **else**
30. **continue** ▷ not vulnerable
31. **end**
32. **end**
33. **if** *regex_validator*($R_{U_1,U_2}[i]$) **then**
34. **continue** ▷ not vulnerable
35. **end**
36. **if** U_1*_content_similarity_validator*($R_{U_1,U_2}[i]$) **then**
37. **if** $R_{U_1,U_2}[i]$ *is HTTP GET* **then**
38. **if** U_2*_content_similarity_validator*($R_{U_1,U_2}[i]$) **then**
39. **continue** ▷ not vulnerable
40. **else**
41. flag vulnerability ▷ vulnerable
42. **continue**
43. **end**
44. **else**
45. flag vulnerability ▷ vulnerable
46. **continue**
47. **end**
48. **else**
49. **continue** ▷ not vulnerable
50. **end**
51. **end**
52. **end**
53. **return** request/response pairs flagged as vulnerable

Algorithm 1: A2CT workflow.

3.3 Crawling Component

As stated at the beginning of Section 3, A2CT relies on the assumption that website elements that are legitimately accessible for a user can be reached by navigating the web application using the presented UI. The purpose of the crawling component is to derive the corresponding HTTP requests and responses for all users included in the test by crawling the web application. Figure 3 illustrates an overview of the crawling component.

The inputs for the crawling component are the same as for the overall workflow (see Figure 2): the configuration file and the authentication and reset scripts. The crawling component consists of two parts: the actual crawler and a proxy. The crawler executes the crawling of the web application under test by interacting with it using HTTP requests and responses. All requests and responses are sent through the proxy (based on *mitmproxy* (mitmproxy, 2024)), which captures and stores them.

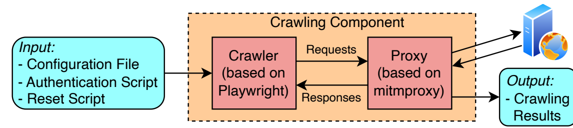


Figure 3: Overview of the crawling component.

In Algorithm 1, the crawling component corresponds to lines 1-5. Crawling is done for each user included in the configuration file and for the *public* user (line 1). Based on the example configuration in Listing 1, crawling is done four times for users *admin*, *seller1*, *seller2* and *public*. For each user, the following steps are done (lines 2-4): First, the web application is reset using the reset script (line 2), which ensures the same initial state before crawling with a user begins. Next, if the current user is not the *public* user, the authentication script is executed within a browser instance using the credentials from the configuration file (line 3). After that, crawling of the web application is done using this authenticated browser instance (line 4). The output of the crawling component are the crawling results, which consist of the crawled HTTP request/response pairs for each user.

The previous version of the crawling component was based on *Scrapy* (Scrapy, 2024) and *Puppeteer* (Puppeteer, 2024). The motivation for using two technologies was to maximize crawling coverage, as depending on the web application, one of them may work better than the other. In reality, however, the usage of Scrapy provided almost no crawling coverage benefits compared to Puppeteer. Therefore, it was decided to stop using the Scrapy framework, and we also replaced Puppeteer with the browser test automation framework *Playwright* (Playwright, 2024a). The latter was mainly driven by the fact that Playwright provides additional flexibility over Puppeteer (e.g., multi programming language and browser support). Other than that, the two technologies, are similar.

Playwright uses a browser instance when crawling a web application. This means that web pages received from the web application are rendered in this browser instance and included JavaScript code is executed. Based on this, Playwright interacts with the UI elements as a user would. As a result, Playwright works well with both traditional and modern web ap-

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

plications and the requests issued during crawling can address, e.g., resources that serve entire HTML pages or resources that correspond to REST API endpoints. This is an important prerequisite for A2CT to detect vulnerabilities in different resource types.

Another important change we did was to integrate user authentication into the browser instance used during crawling. In the previous version, authentication was done separately and the resulting authentication information such as cookies or tokens was added to requests on the fly by the proxy during crawling, but this was not always reliable and did not work with every web application. By executing the authentication script directly within the browser instance that is used for crawling afterwards, we get a truly authenticated browser instance that correctly stores cookies or tokens, and as a result of this, they are used correctly during crawling. This integration also makes sure that cross-site request forgery (CSRF) countermeasures such as CSRF tokens are handled correctly, which is an important prerequisite when supporting request types other than GET. As a final note, it is important that no accidental logout happen during crawling by clicking on a corresponding button or link, but this can usually be avoided by blacklisted keywords, which has the effect that requests where the URL contains such a keyword are not executed during crawling (see *do_not_call_pages* option in Listing 1).

In addition, the actual crawling process was also significantly redesigned and improved as with the previous version, we often observed limitations such as UI elements that were not detected (and therefore web application areas that could no be reached) and infinite crawling loops. In the following, some insights into how the crawling component works are provided, with a focus on the recent enhancements.

At the beginning of the crawling process, the main page of the web application is requested and all navigation elements are identified. We distinguish between *link elements* and *clickable elements*. For instance, anchor tags (<*a*>) with an *href* attribute that contains a URL are link elements that load a new page. Examples of clickable elements are <*button*> tags or *input[type=submit]* elements in forms. The detected elements are converted to tasks and added to a first in, first out (FIFO) task queue. After this initial step, the tasks in the queue are processed. If the task contains a link element, the corresponding resource is requested and processed in the same way as with the main page. If the task contains a clickable element, processing is more complicated, as interacting with such an element typically leads to one of two outcomes: (1) changes in the document object model (DOM) of the current page (e.g., new clickable elements are created, become actionable, change or even disappear) or (2) a page navigation is initiated, which loads a new page. The crawling component can detect these two cases and reacts accordingly: If a page navigation happens, the newly loaded page is immediately processed, just like the main page. If no navigation is detected, the state of the DOM is compared with the previous state to detect new UI elements that may have appeared. All newly detected link elements and clickable elements are added to the task queue. With this process, the crawling component gradually explores the web application as it continuously processes newly detected pages, follows link elements, and interacts with clickable elements by processing tasks from the task queue until it is empty.

Some clickable elements only become actionable after other clickable elements have been clicked first. For instance, there may be a navigation menu where the buttons use multiple hierarchy levels where the next next level is only shown after the button corresponding to the parent menu entry has been clicked. The crawling component supports this by implementing a recursive clicking approach. This is done by performing a series of clicks on already visible clickable elements, which allows to access UI elements that are only reached after a certain clicking depth. Compared to the previous version, this newly introduced recursive clicking approach was a major step to improve crawling coverage in modern web applications as it allows to detect significantly more UI elements and therefore also web application areas.

Typically, the same elements will be found multiple times during crawling. Therefore, for efficiency reasons and to prevent infinite crawling loops, an element is only added to the task queue if it is considered new. This is done by taking the *CSS Path* as well as the *outerHTML* property of a UI element into consideration, as these two values uniquely identify a UI element in a web application. A newly detected UI element is only added to the task queue if no task with the same two values is already existing in the task queue or has been processed before.

3.4 Filtering Component

The filtering component gets the configuration file and the crawling results as inputs and uses five sequential filters to remove HTTP request/response pairs from the crawling results that are unnecessary and to prepare the input for the following replaying component. Figure 4 shows the filtering component.

In Algorithm 1, the filtering component corresponds to lines 6-14. The first four filters are applied to the crawling results per user (line 6). The first filter,

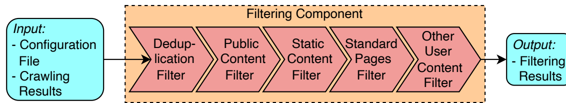


Figure 4: Overview of the filtering component.

the *deduplication filter* (line 7), removes duplicate request/response pairs from the crawling results. In the previous version, duplicate checks were integrated in the proxy of the crawling component before storing a request/response pair. However, this had the consequence that the complete crawling results were never stored, which made it difficult to understand and improve the crawling component during development. Also, the complete crawling results can be helpful to understand why a specific vulnerability was reported by A2CT, as they allow to reconstruct the sequence of requests that led to this result, and the final validator of the validation component (see Section 3.6) works most reliably if the complete crawling results are available. The deduplication filter goes through the crawled request/response pairs and removes an entry if the request is exactly the same or similar to a request that has been observed before. Specifically, two requests are considered to be similar if they only differ in the value of a CSRF token or in the ordering of parameter/value pairs (or key/value pairs with JSON data) in the query string or request body.

The second filter, the *public content filter* (line 8), removes entries from the crawling results that were also found when crawling with the *public* user. As these resources could be reached without logging in, they are considered public content and not relevant in the context of access control vulnerabilities.

The third filter, the *static content filter* (line 9), removes entries that correspond to static content. Typical static content includes, e.g., JavaScript files, CSS files, font files and – depending on the web application – image files. This filter can be configured by specifying the corresponding file extensions (*static_content_extensions* option in Listing 1).

The fourth filter, the *standard pages filter* (line 10), removes entries that match certain standard pages. Standard pages may include "About Us" pages, contact forms, etc., that are not relevant for access control tests. This filter can be configured by specifying the corresponding resources (*standard_pages* option in Listing 1).

The fifth filter, the *other user content filter* (lines 12-14), is applied in the context of each user pair that is included in the test. As an example, consider user pair *admin seller1* from Listing 1. Assume that F_{admin} and $F_{seller1}$ contain the remaining request/response pairs after the first four filtering stages (line 10). In this case, the request/response pairs in F_{admin} are taken as a basis and all entries are removed from this basis if the request is also included in $F_{seller1}$, because these resources can – according to the fundamental assumption – legitimately be accessed (by crawling) by both users. The remaining request/response pairs (identified as $F_{admin,seller1}$, line 13), however, are candidates for access control vulnerabilities if the resources can successfully be accessed by *seller1*. As an example, consider Figure 5.

Figure 5: Example of the other user content filter.

In this example, after the first four filtering stages, four entries are remaining in F_{admin} and three in $F_{seller1}$ (green boxes). For illustrative purposes, only the request types and the URLs are shown here, but in reality, the full request/response pairs are stored. In $F_{admin,seller1}$ (purple box), only the two red requests from F_{admin} are remaining, because the two blue requests from F_{admin} are also in $F_{seller1}$ and are therefore not included in $F_{admin,seller1}$. The same is done for the other three user pairs configured in Listing 1.

The output of the filtering component are the filtering results, which consist of the filtered HTTP request/response pairs for each configured user pair.

3.5 Replaying Component

The replaying component takes the configuration file, the authentication and reset scripts, and the filtering results as inputs and replays each request in the filtering results using the appropriate user identity. Figure 6 illustrates the replaying component.

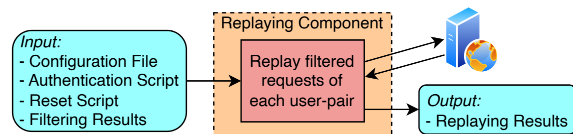


Figure 6: Overview of the replaying component.

In Algorithm 1, the replaying component corresponds to lines 15-19. For each entry in the *user_pairs* node in the configuration file (line 15), the following steps are performed: First, the web application is reset using the reset script (line 16), which ensures the same initial state before replaying with a user begins. If the second user in the current *user_pairs* entry is not the *public* user, the authentication script is executed next (line 17), using the credentials of this second user. This is done in the same way as in the crawling component, i.e., directly within a browser instance.

Next, the requests from the filtering results of the current user pair are replayed (line 18). This is done by taking the original request from the filtering results, replacing authentication information (e.g., cookies or tokens) in the original request with the fresh information that is extracted from the authenticated browser instance, sending the updated request to the web application, and receiving the response. Recalling the example in Figure 5 for user pair *admin seller1*, this means that the two red requests in the purple box would be replayed using the identity of *seller1*.

The output of the replaying component are the replaying results, which consist of the replayed HTTP request/response pairs for each configured user pair.

3.6 Validation Component

The validation component takes the configuration file and the replaying and crawling results as inputs and uses a series of validators to determine which replayed HTTP request/response pairs correspond to access control vulnerabilities. Figure 7 illustrates the validation component. At the top, a high-level overview is shown, and at the bottom, the details how the validators are applied to one request/response pair from the replaying results are visualized.

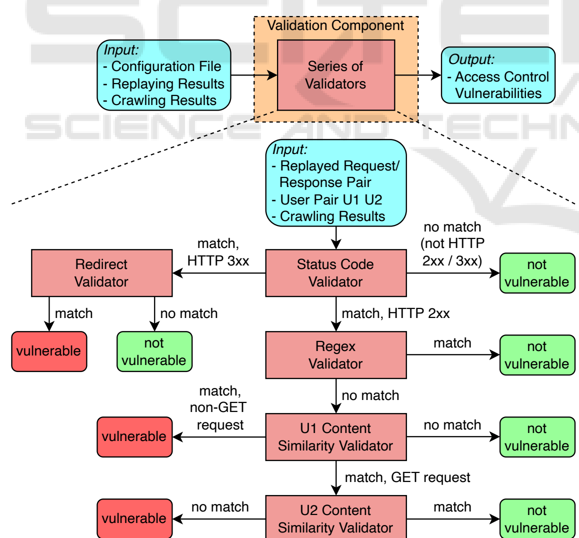


Figure 7: Overview of the validation component (top) and series of validators (bottom).

In Algorithm 1, the validation component corresponds to lines 20-53. In the following, we focus on the bottom part of Figure 7. The series of validators is applied to each user pair U_1 U_2 defined in the *user_pairs* node in the configuration file (line 20) and to each request/response pair from the replaying results of this user pair (line 21). Recalling the example in Figure 5 for user pair *admin seller1*, then U_1 corresponds to *admin*, U_2 corresponds to *seller1*, and the series of validators is applied separately to the two request/response pairs in the purple box.

First, the *status code validator* (line 22) checks whether the HTTP status code of the response of the replayed request indicates a successful access. Status codes between 200-302 and status code 307 are considered potentially successful accesses, whereas any other status code (e.g., 4xx and 5xx) is considered an access denied decision by the web application, which rules out the possibility of an access control vulnerability (line 23). Note that a 2xx status code alone does not yet confirm that access was permitted since, e.g., the content of the response may show a custom error message that access is not allowed (instead of responding, e.g., with a 403 status code).

In case of a redirection response (i.e., status code 3xx, line 25), the *redirect validator* is invoked next (line 26). It compares the *Location* response header in the replayed response of U_2 with the *Location* header in the corresponding response in the crawling results of U_1. If they are equal, the request/response pair is flagged as vulnerable (line 27). If they are different, no vulnerability is reported (line 30). The rationale here is that disallowed accesses are often redirected to an error page or the login page, so they can be distinguished from permitted accesses by different URLs in the *Location* header. As an example, consider the POST request in the purple box of Figure 5. Assume that during crawling with *admin*, a redirection response to https://site.com/admin/showusers was received. If the same redirection response with the same URL is received when replaying this request as *seller1* (i.e., the *Location* headers match), then it is very likely a vulnerability has been detected. On the other hand, if a redirection response to, e.g., https://site.com/error or https://site.com/login is received (i.e., the *Location* headers differ), then it is very likely that there is no vulnerability in the context of this request/response pair.

In case of a response with status code 2xx, the *regex validator* (line 33) is used after the *status code validator*. It simply checks if the response body matches a specific regular expression. This regular expression can optionally be configured with the *regex_to_match* option in Listing 1. Its purpose is to easily recognize specific strings (e.g., custom access denied error messages) in the response body that indicate a blocked access. If a match is detected, the response of the replayed request is considered as a denied access and no vulnerability is flagged (line 34).

Next, the U_1 *content similarity validator* (line 36) checks if the response body in the replayed response

of U_2 is similar to the corresponding response body in the crawling results of U_1. If they are significantly different, no vulnerability is flagged (line 49) as the web application very likely did not send back information that should be inaccessible by U_2. On the other hand, if the response bodies are similar, a vulnerability may have been detected as U_2 managed to access similar content as U_1. For details about how the similarity of the response bodies (e.g., HTML documents or JSON data) is determined, we refer to our previous paper (Rennhard et al., 2022), but the basic idea is that with HTML pages, the visible textual elements are extracted and compared, and with JSON data, the key/value pairs are compared. If there is an overlap of 80% or more, then the response bodies are considered similar. As an example, consider the GET request in the purple box of Figure 5. Assume that during crawling with *admin*, an HTML page with the admin dashboard is received. If the same or almost the same dashboard is received when replaying the request as U_2, then the responses are considered similar and a potential vulnerability has been detected. Conversely, if a significantly different response is received by U_2, e.g., because it contains an access denied message or only a skeleton HTML page without the actual dashboard, the responses are not considered similar and no vulnerability is flagged.

If the previous validator identified a potential vulnerability and if the request is a GET request (line 37), the U_2 *content similarity validator* (line 38) is invoked as a final check. It compares the response body in the replayed response of U_2 with all response bodies in the crawling results of U_2. If any of these response bodies is similar to the one of the replayed response, then no vulnerability is flagged (line 39), otherwise a vulnerability is flagged (line 41). The reasoning here is that if U_2 can access the content of the replayed response also by crawling (via a different URL), then U_2 apparently has legitimate access to this content and consequently, no vulnerability should be flagged. The main purpose of this validator is to reduce false positives. To illustrate this, assume that during crawling, U_1 and U_2 use different URLs to access the same content, e.g., because the URL paths or a sorting parameter in the URL differ. In such cases, if we assume that U_2 can indeed access the same content also with the corresponding URL found during crawling with U_1, the U_1 *content similarity validator* flags this as a potential vulnerability, i.e., a false positive. With the U_2 *content similarity validator*, however, this is recognized and no vulnerability is reported.

Note that with non-GET requests (line 44), this final validator is not used and a vulnerability is flagged (line 45). To illustrate this, assume that both U_1 and U_2 have legitimate access to the list of customers in a web shop and can find this resource during crawling. In addition, U_1 (but not U_2) is also permitted to delete a customer using a POST request, and as a response to this request, U_1 gets the updated list of customers. Now assume there is a vulnerability in the sense that U_2 can also delete a customer by issuing the same POST request and also getting the same response as U_1. The U_1 *content similarity validator* flags this correctly as a potential vulnerability. However, if the received response – the updated list of customers – were now compared with the crawling results of U_2 using the U_2 *content similarity validator*, it would be very likely that no vulnerability would be flagged as U_2 received a very similar list of customers during crawling. To prevent such cases, the U_2 *content similarity validator* is only used with GET requests.

The output of the validation component are the detected access control vulnerabilities, i.e., the list of request/response pairs and the corresponding user pairs that were flagged as vulnerable (line 53).

4 EVALUATION

In the evaluation of the previous version, we used only a small set of web applications where we added specific vulnerabilities to show that the approach works in principle. With the evaluation presented here, we want to demonstrate that A2CT can indeed detect previously unknown access control vulnerabilities to prove its true value in practice.

To do this, A2CT was applied to several publicly available open-source web applications. Overall, 30 web applications were tested and A2CT managed to find 14 vulnerabilities in two of them. These vulnerabilities resulted in six newly published CVE records. Table 1 gives an overview of the vulnerabilities, including a number, the web application, the affected version, the access control vulnerability type, the CVE record, and the HTTP method and URL of the vulnerable request. The remainder of this section describes the vulnerabilities in detail.

4.1 Unifiedtransform

Unifiedtransform (Unifiedtransform, 2024) is an open-source school management and accounting software. Seven vulnerabilities were detected:

- **No. 1: A student can view the grades of other students.** An object-level access control vulnerability, which can be exploited by setting the URL parameter *student_id* to the ID of another student.

Table 1: Detected vulnerabilities.

No.	Web Application	Version	Access Control Vulnerability Type	CVE Record	HTTP Method	URL path and parameters
1	Unifiedtransform	≤2.0	Object-level	CVE-2024-12305	GET	/marks/view?student_id=<student_id>&course_id=...
2			Function-level	CVE-2024-12306	GET	/students/view/list
3			Object-level	CVE-2024-12306	GET	/students/view/profile/<student_id>
4			Function-level	CVE-2024-12306	GET	/teachers/view/list
5			Function-level	CVE-2024-12306	GET	/teachers/view/profile/<teacher_id>
6			Function-level	CVE-2024-12307	GET	/students/edit/<student_id>
7			Function-level	CVE-2024-12307	POST	/school/student/update
8	Mautic	≤4.4.9	Object-level	CVE-2024-2730	GET	/page/preview/<incremented_number>
9			Function-level & SSRF	CVE-2024-3448	GET	/s/ajax?action=plugin:focus:checkIframeAvailability&...
10			Function-level	CVE-2024-2731	GET	/s/contacts/quickAdd
11			Function-level	CVE-2024-2731	GET	/s/contacts/batchOwners
12			Function-level	CVE-2024-2731	GET	/s/monitoring
13			Function-level	CVE-2024-2731	GET	/s/companies/merge/<company_id>
14			Function-level	CVE-2024-2731	POST	/s/tags/edit/<tag_id>

- **No. 2,3: A student can view personal information of other students.** A function-level access control vulnerability on page */students/view/list* allows students to list all other students. Also, detailed personal information can be accessed with */students/view/profile/<student_id>*, which is an object-level access control vulnerability.
- **No. 4,5: A student can view personal information of teachers.** Similar to No. 2,3, students can access */teachers/view/list* to get the list of all teachers and */teachers/view/profile/<teacher_id>* to reveal personal information of them.
- **No. 6,7: A teacher can edit student data.** A function-level access control vulnerability allows teachers to illegitimately access */students/edit/<student_id>*. On this page, student data such as name, date of birth, address, etc., can be modified, which sends a POST request to */school/student/update* to illegitimately save the changes.

4.2 Mautic

Mautic (Mautic, 2024) is an open-source marketing automation software used by more than 200'000 organizations. Seven vulnerabilities were uncovered:

- **No. 8: Previews of marketing campaigns can be accessed by non-authenticated users.** Marketing campaigns created in Mautic can be published as a preview, which can be accessed by a public URL. The idea is that only recipients of the URL can access the preview. A2CT reported that this URL can illegitimately accessed by the public user and further analysis revealed that guessing valid URLs is easy, as the number to identify a preview is assigned incrementally. This object-level access control vulnerability allows competitors to learn about upcoming campaigns and products, which can provide a competitive advantage.
- **No. 9: Improper access control leads to Server-Side Request Forgery (SSRF).** This is a function-level access control vulnerability where A2CT found that the AJAX function identified with parameter *action=plugin:focus:checkIframeAvailability* can be executed by low-privileged users. This can be used to load any HTML page within an iFrame. Further investigation revealed that this can be abused to carry out an SSRF attack to execute, e.g., internal and external port scans originating from the server where the Mautic instance is running.
- **No. 10-14: Low-privileged users can view pages that expose sensitive information:** These are function-level vulnerabilities, which are all similar. */s/contacts/quickAdd* exposes all users' full names, all company names, all stage names, and all tags (the latter two are labels that can be assigned to contacts); */s/contacts/batchOwners* exposes all users' full names; */s/monitoring* exposes all monitoring campaigns and their descriptions; */s/companies/merge/<company_id>* exposes all company names; and */s/tags/edit/<tag_id>* exposes tag names and descriptions and allows descriptions to be edited.

4.3 Summary

The 14 vulnerabilities show the versatility of A2CT, as they cover reading (GET) and modifying (POST) requests, requests for HTML documents and to APIs, and function- and object-level vulnerabilities.

Beyond the uncovered vulnerabilities, A2CT also reported some false positives. Usually, they can be quickly identified – in particular by a person who has a good understanding of the web application – by visually inspecting the requests, or by manually interacting with the web application, sending the corresponding requests, and checking the responses. In one case, however, we were convinced that we detected two vulnerabilities that allowed so-called worker users access to administrator functions, and we reported this to the project maintainers. They

wrote us back that these are not vulnerabilities, but at the same time they thanked us as our findings revealed that there are functional bugs in their web applications in the sense that the functionality to access these administrative functions was missing in the UI presented to worker users. This shows that while A2CT is designed to detect access control vulnerabilities, it can also uncover other issues in a web application.

5 DISCUSSION

Compared to our previous version, A2CT is a big step forward. It removes the previous limitations in the sense that it now supports all HTTP request types and not only GET requests, that it copes much better with the wide range of web application types (based on traditional and modern architectures), that testing coverage and robustness were significantly improved by redesigning the crawling component, and that multiple user combinations can be tested in one test run.

Beyond these technical improvements, the most important step forward is the evaluation scope. Before, the evaluation was based solely on a small set of web applications where we deliberately added vulnerabilities. With the evaluation presented here, we have proven that A2CT can not only be applied to a wide range of web applications, but that it can indeed find previously unknown access control vulnerabilities. This demonstrates the practical value of A2CT.

Some limitations remain. If the fundamental assumption (see beginning of Section 3) on which A2CT relies is not valid in a web application, then A2CT will not work well and many false positives may be reported. However, our evaluation demonstrated that in most web applications, this assumption is valid. Another limitation is that supporting modifying HTTP requests during crawling can cause state changes in the web application, which may affect the results. This was partly remedied by resetting the web application whenever crawling with a new user begins, but state changes while crawling with a specific user can of course still have an impact.

6 RELATED WORK

One way to find access control vulnerabilities in web applications is to automatically extract the access control model, so it can be verified. In (Alalfi et al., 2012), the source code of a web application is analyzed to create a role-based access control model, which is then manually checked for correctness. The disadvantages of this approach are that it is dependent on the used programming language and/or web framework, meaning it has to be adapted for different technologies, and that manual checking is required. In (Le et al., 2015), a web application is crawled with different users to create access spaces per user. Next, a machine learning-based approach is used to derive access rules from these access spaces. These rules are then automatically compared against an existing access control specification or manually assessed. While this approach does not require access to source code, it still requires an existing access control model for the approach to be automated.

Several approaches use replay-based vulnerability detection. In (Li et al., 2014), an access control model is extracted from a web application by crawling it with different users. During crawling, database accesses by the web application are monitored to derive an access control model that describes relationships between users and permitted data accesses. This model is then used to create test cases that check whether users can access data that should not be accessible by them. In (Noseevich and Petukhov, 2011), typical sequences of requests in web applications are compared with random sequences created by crawlers. For this, a human first browses a web application manually to record typical use cases, which are represented as 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 uncovered several vulnerabilities. In (Xu et al., 2015), an approach is described where a role-based access control model of a program must first be defined manually. This model is then used to create access control test cases, which are transformed to code to execute the tests. This approach was applied to three Java programs, where it demonstrated that it could automatically create a significant portion of the required test code and that it could uncover vulnerabilities.

Some recent approaches focus on automated REST API testing (Atlidakis et al., 2019; Viglianisi et al., 2020; Liu et al., 2022; Deng et al., 2023). Based on the OpenAPI specification, they generate request sequences with the goal of triggering and detecting bugs in the API. Although none of the works focus solely on access control, *Nautilus* (Deng et al., 2023) uncovered some privilege escalation vulnerabilities.

7 CONCLUSION

In this paper, we presented A2CT, a practical approach for the automated detection of function- and

object-level access control vulnerabilities in web applications. A2CT can be applied to a wide range of web applications, requires only a small configuration effort, and can detect vulnerabilities in the context of all relevant HTTP request types (GET, POST, PUT, PATCH, DELETE). During our evaluation in the context of 30 publicly available web applications, we managed to detect 14 previously unknown vulnerabilities in two of these applications, which resulted in six newly published CVE records. This demonstrates the soundness of the solution approach and the practical applicability of A2CT.

To our knowledge, A2CT is the first approach that combines a high degree of automation, broad applicability, and demonstrated capabilities to detect access control vulnerabilities in practice. To encourage further research in this direction, A2CT is made available to the community under an open-source license.

ACKNOWLEDGEMENTS

This work was partly funded by the Swiss Confederation's innovation promotion agency Innosuissse (project 48528.1 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.

Atlidakis, V., Godefroid, P., and Polishchuk, M. (2019). RESTler: Stateful REST API Fuzzing. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 748–758.

Bennets, S. (2023). Open Source Web Scanners. https://github.com/psiinon/open-source-web-scanners.

Deng, G., Zhang, Z., Li, Y., Liu, Y., Zhang, T., Liu, Y., Yu, G., and Wang, D. (2023). NAUTILUS: Automated RESTful API Vulnerability Detection. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 5593–5609, Anaheim, CA. USENIX Association.

Docker (2024). Docker. https://www.docker.com.

Kushnir, M., Favre, O., Rennhard, M., Esposito, D., and Zahnd, V. (2021). 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*, pages 204–216.

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.

Liu, Y., Li, Y., Deng, G., Liu, Y., Wan, R., Wu, R., Ji, D., Xu, S., and Bao, M. (2022). Morest: Model-based RESTful API Testing with Execution Feedback. ICSE '22, page 1406–1417, New York, NY, USA. Association for Computing Machinery.

Mautic (2024). Mautic. https://www.mautic.org.

mitmproxy (2024). mitmproxy. https://mitmproxy.org.

MITRE (2024). CVE – Common Vulnerabilities and Exposures. https://www.cve.org.

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

OWASP (2021a). OWASP Top 10 – A01:2021 – Broken Access Control. https://owasp.org/Top10/A01_2021-Broken_Access_Control.

OWASP (2021b). OWASP Top 10:2021. https://owasp.org/Top10.

OWASP (2024). Vulnerability Scanning Tools. https://owasp.org/www-community/Vulnerability_Scanning_Tools.

Playwright (2024a). Playwright. https://playwright.dev.

Playwright (2024b). Playwright Codgen. https://playwright.dev/docs/codegen##running-codegen.

Portswigger (2024). Access Control Vulnerabilities and Privilege Escalation. https://portswigger.net/web-security/access-control.

Puppeteer (2024). Puppeteer. https://pptr.dev.

Rennhard, M., Kushnir, M., Favre, O., Esposito, D., and Zahnd, V. (2022). Automating the Detection of Access Control Vulnerabilities in Web Applications. *SN Computer Science*, 3(5):376.

Scrapy (2024). Scrapy. https://scrapy.org.

SQLite (2024). SQLite. https://www.sqlite.org.

Unifiedtransform (2024). Unifiedtransform. https://changeweb.github.io/Unifiedtransform.

Verizon (2023). 2023 Data Breach Investigation Report. https://www.verizon.com/business/en-gb/resources/reports/dbir.

Viglianisi, E., Dallago, M., and Ceccato, M. (2020). RESTTESTGEN: Automated Black-Box Testing of RESTful APIs. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, pages 142–152.

Xu, D., Kent, M., Thomas, L., Mouelhi, T., and Traon, Y. L. (2015). Automated Model-Based Testing of Role-Based Access Control Using Predicate/Transition Nets. *IEEE Transactions on Computers*, 64(9):2490–2505.

ZHAW Infosec Research Group (2024). A2CT. https://github.com/ZHAW-Infosec-Research-Group/A2CT.