Evolutionary Search for Authorization Vulnerabilities in Web Applications

by

Akshay Kawlay

A thesis submitted in conformity with the requirements
for the degree of Master of Applied Science
Graduate Department of Electrical and Computer Engineering
University of Toronto

Abstract

Evolutionary Search for Authorization Vulnerabilities in Web Applications

Akshay Kawlay
Master of Applied Science
Graduate Department of Electrical and Computer Engineering
University of Toronto
2021

Proper access controls are essential to protect private resources in web applications. However, authorization vulnerabilities resulting from broken access controls have been common in the past. Existing techniques to detect such vulnerabilities require manual effort and suffer from false positives as automating authorization vulnerability detection in an app agnostic way is challenging.

We present two subproblems - automated discovery of app resources and automated detection of authorization vulnerabilities. This thesis attempts to solve these subproblems by introducing AuthZee, a tool designed to automatically discover resources in a web app and automatically detect if those resources are vulnerable to improper authorization, without requiring details about the app logic. We propose a novel way of crawling using an evolutionary algorithm to generate app objects and triad testing for detection. AuthZee requires login credentials for three user accounts in the target app which allows it to crawl user account space and perform triad testing.

Upon testing with 7 popular open source web applications diverse in size and logic, we find that our approach is able to discover more resources than existing crawling techniques and perform detection with 0 false positives in most apps.

Acknowledgements

I'm very grateful for the support and guidance I received from my supervisor, Professor David Lie. His patient approach and invaluable feedback allowed me to explore many unconventional ideas and incrementally improve my solution throughout the course of my Master's degree.

I am also grateful for the financial support from the Department of Electrical and Computer Engineering of the University of Toronto

I would like to thank Truman Jian for helping me run experiments and gather data for my project.

I would like to extend my thanks to the online community on platforms like stackoverflow, github, especially testcafe developers for answering my questions in a relatively short time.

Lastly, I would also like to appreciate my family for always believing in me and supporting me throughout my life.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

More and more important services are provided via web applications today. The transition of services like banking, government, businesses, education, entertainment and e-commerce to the web has greatly improved their quality and accessibility, but has also exposed them to adversaries who may seek to access private information or perform unauthorized actions. Web apps can fall victim to such adversaries due to mismanaged access controls, missing or improper authorization checks or software defects in the web app logic, allowing adversaries to perform unauthorized actions on the web application like accessing somebody else's private resources. Such software defects are called *authorization vulnerabilities* and the resources exposed are said to be vulnerable to unauthorized access. Such vulnerabilities have allowed malicious adversaries to compromise and access millions of user accounts in vulnerable web applications in recent years [46] [40].

Authorization vulnerabilities have been growing year by year and have been in OWASP's top 10 most common vulnerabilities for the last four years [16] [19] [17]. Of these vulnerabilities, IDOR which stands for Insecure Direct Object Reference, are very common. Bug bounty platforms like Hackerone report over 200 IDOR vulnerabilities every month [28]. One of the main reasons for this is that authorization rules and user permissions are often specific to the web application logic. This results in developers having to manually craft bespoke user permissions and authorization rules into the web application logic. For a reasonably complex application, the number of access rules increases quickly and becomes difficult to maintain in the face of frequently changing designs and requirements [34] [23].

There exist static analysis tools that can detect authorization vulnerabilities [55]. However, because such tools do not actually execute the applications, they are limited by static analysis imprecision and thus may suffer from a large number of false positives or false negatives. Dynamic analysis approaches that find vulnerabilities by executing the web applications can offer more precise results, but require more manual effort from the human tester. One such approach is manual testing—in the simplest case, the tester human logs in with two different user credentials. Then the tester attempts to visit a user's resource URL from a different user's account and visually verifies that the response to a request does not contain data of another user or the unauthorized request to fetch, modify or create an object is unsuccessful. Despite its simplicity, some variant of this basic manual approach has been behind the discovery of IDOR vulnerabilities in web apps such as Shopify [37], Twitter [44], New Relic [38], the US DoD [35] and Crowdsignal [26]. These instances of IDORs are typically found manually using a web

proxy interceptor, repeater like Burp [22], ZAP [14]. There exist semi-automated techniques however human effort is still required for detection.

Due to the diversity of web application user interfaces (UI), it is difficult to automatically determine whether an access to a resource URL is successful or not. For example, many applications do not properly set the HTTP response code to indicate whether a request is successful or not. While web applications are supposed to return a 200 code on successful requests and a 4xx or 5xx code on an error, many applications do not obey this convention. Instead, they may return a 200 code, but indicate in the web page HTML that the request was not successful. As a result, HTTP response codes cannot be used to reliably determine whether an access is successful or not and thus determine whether an unauthorized user is able to access a resource they should not have access to.

Another challenge is to discover and specify the request URLs and API endpoints through which resources can be accessed. For example, while tools such as Autorize [20], an extension to the Burp Suite [6], which automatically repeats every request made by a user in another user's session can automate the driving of test inputs, but still requires a human to specify the requests or APIs to interact with the web application. Similarly, AuthMatrix [4] takes a table of users, their roles, and requests and then tries all combinations to detect if some combination results in a vulnerability. However, the requests must still be manually specified in the table given to AuthMatrix. To get these requests and APIs, previous work, such as Fuzz-lightyear [18] and Restler [24] have relied on OpenAPI specifications, but such specifications are not ubiqutously available. Automatically finding the request URLs also has a number of challenges. Existing approaches like website scanners and crawlers can discover new pages and requests, but they are limited in two ways. First, they interact directly with the web application and are not able to interpret Javascript, limiting their ability to crawl web applications that use AJAX requests or dynamically generated Javascript content and resources [33] [29], both of which are common on complex web application where authorization vulnerabilities exist. Second, existing crawlers and scanners are often not able to interact with web applications to submit forms or create new objects on the web application. For example, they typically cannot create a new submission on a conference submission application like HotCRP, or create a new Git repository on a Git management application like GitLab. Without the ability to interact with the web application, they will most likely only discover publicly accessible web pages, which can't be vulnerable to authorization vulnerabilities since they are not access controlled. This can lead to false positives—for instance, Fuzz-lightyear needs a tester to manually specify the publicly accessible resources to ignore during testing, otherwise it may mistake a successful access to a public page for a vulnerability that allows an unauthorized access to a private resource.

In summary, there are two challenges to improving the effectiveness of automated authorization vulnerability detection in web applications. First, since every web application is different, we must automatically discover the URLs that are used to access protected objects in the web application. Second, we must be able to automatically detect if an unauthorized access was allowed to a web object via the discovered URLs. To overcome these two challenges, we propose the first automated tool for object generation and authorization testing designed to find IDOR vulnerabilities - AuthZee, in an app agnostic manner. AuthZee uses a crawler enhanced with an evolutionary search algorithm to interact with web applications to generate and discover web object URLs. The thesis introduces a search problem

to find sequences of interactions on the app UI elements that sends requests to app server to generate new objects. The Evolutionary Algorithm (EA) empowers the crawler to search for such multi step interactions on the app UI. This allows AuthZee to both automatically create and access private objects on web applications. To detect unauthorized accesses, AuthZee uses a novel *Triad test*, which compares the HTML or JSON the web application responds with among three users in a web application-agnostic way.

1.1 Contributions

This thesis makes following contributions in the field of web authorization vulnerability detection:

- A novel crawling technique leveraging an evolutionary algorithm to generate objects and discover web app resources. A custom browser automation tool built on testcafe framework, is developed to implement this technique.
- Triad detection - an app agnostic way to automatically test for authorization vulnerabilities.
- An evaluation of our crawling and detection techniques showing that they discover deeper app resources and successfully detect if a resource is vulnerable to unauthorized access

1.2 Thesis Structure

The thesis provides the necessary background on concepts such as Authorization, IDOR vulnerabilities, Evolutionary Algorithms and Gestalt Pattern Matching in Chapter 2. Following that, the design of AuthZee is described in Chapter 3. Next, Chapter 4 explains the implementation of AuthZee along with additional requirements needed. Subsequently, Chapter 5 presents various experiments run to evaluate the performance. Furthermore, the limitations learnt are explained in Chapter 6 and a literature review of related work is conducted in Chapter 7. Finally, Chapter 8 presents the future work and concludes this thesis.

Chapter 2

Background

This section explains the concepts necessary to understand our design and the authorization vulnerabilities it is designed to detect. We start by differentiating between authentication and authorization, followed by describing the types of broken access controls. We then mention basic client-server architecture of web applications and how browser automation powers end to end testing. Finally, we briefly explain evolutionary algorithm concepts.

2.1 Access Control and Authorization

It is important to distinguish between authentication and authorization. Authentication (AuthC) is the process where the web server attempts to identify the user whereas authorization (AuthZ) is the process where the web server may already know who the user is and then determines if that user should be allowed to perform the action they requested. Authorization in a web application is modeled by its access control rules. There are three types of access controls - Horizontal, Vertical and Context-dependent [52]. Horizontal access controls restrict access to functionality within the same privilege level. For example, a user would be restricted viewing/modifying another user account's private details. Similarly, vertical access controls restrict access to functionality needing a higher privilege level. For example, a regular user would be restricted from accessing admin pages. On a different note, the context-dependent access controls restrict access to functionality based on the application state as a result of user interactions. For instance, a user in an ecommerce application would be restricted from updating her transaction details after that transaction was completed.

Authorization vulnerabilities can result from poor implementation or mismanagement of the application's access controls. Following are the examples of broken access controls that this thesis focuses on - Vertical privilege escalation - user gains access to resources belonging to a higher privileged user or admin; Horizontal privilege escalation - user gains access to another user's resources at same privilege; Horizontal to vertical privilege escalation - a horizontal privilege escalation attempt leads to gain of higher privilege;

IDOR is a common type of authorization vulnerability that results from broken access controls. It stands for Insecure Direct Object Reference where modifying the input parameters to a resource (URL) allows direct access to unauthorized content [52]. For instance, a URL of the form :

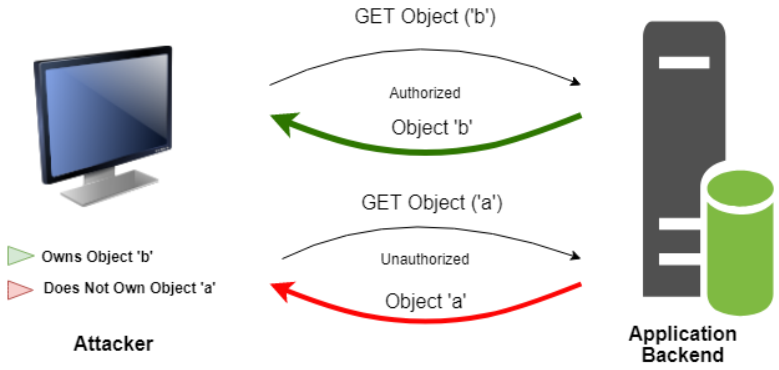

Figure 2.1: Illustrating Insecure Direct Object Reference

`http://www.website.com/some-object-path?param1=val1&param2=val2`

takes two input parameters - param1 and param2, set to val1 and val2 respectively. Visiting this URL would send a request for some object in the app. Thus, the URL is a reference to that object. An attacker looking for IDORs may change the input parameter values i.e val1 and val2, such that the resultant URL references some other object belonging to another user. An app with an IDOR vulnerability would comply with the attacker's crafted request to access another user's objects.

2.1.1 Examples of IDOR vulnerabilities

IDORs are mostly manually detected since automated blackbox techniques like web vulnerability scanners cannot detect them. There are two main manual methods to perform an IDOR attack. First, parameter tampering, where a request is sent with modified input parameters that point to another private object. Here, guessing the correct input parameters (objectIDs) that actually refer to an object can be challenging, especially if the app uses randomized objectIDs. Second, cookie swapping, where the request is replayed with the original legitimate input parameters but the session cookies in the request header are modified to an unauthorized user's session cookies. This method requires a web proxy to intercept the request, update the cookies and resend. Furthermore, IDORs in object creation requests (POST request) would also require a web proxy. Both the methods essentially test if an unauthorized user can access a resource. IDORs can impact applications in many different ways. We describe some past IDOR vulnerabilities reported on the bug bounty platforms.

Crowdsignal app provides services to create customized surveys, polls and quizzes. Last year, an IDOR vulnerability in their invite-user.php resource -

`https://app.crowdsignal.com/users/invite-user.php?id=(userid)&popup=1`

allowed the attacker to send the request with a victim's userid resulting in receiving a response with the victim's email and subsequently leading to account takeover after clicking on the 'update Permissions' button on the response page [26]. This example showcases how an IDOR can be used to trigger other vulnerabilities, causing a higher severity impact.

Another IDOR was reported to Twitter 7 months ago. This was found in a create request in Revue

Figure 2.2: Structure of a typical request(left) and response(right)

(getrevue.co). Revue is an editorial newsletter tool built by Twitter for writers, publishers to create newsletters and get paid. An attacker was able to add images, descriptions and titles to other user's issues. A issue in Revue is a weekly or monthly newsletter, in other words a user object in the application. The vulnerability was reported in the endpoint to create images/descriptions/titles in an issue. The attacker could create images/descriptions in the victim's issue by sending a POST request with the issue id modified to the victim's issue id. The URL endpoint hasn't been disclosed yet [44].

IDOR attacks can also allow attackers to bypass the app business logic. For instance, in bWapp, an intentionally vulnerable web app, the business logic can be bypassed in a movie ticket booking platform. Here, an attacker, using a web proxy interceptor, sends a crafted request with the 'ticket_price' input parameter changed to zero, thus allowing her to book tickets for free. This is explained in [27]. In this example, although the attacker was authorized to access the ticket booking URL, the improper logic implementation allowed the attacker to successfully exercise an IDOR attack.

2.2 Web Applications and Browser Automation

Modern web applications can be broadly split into two components - the frontend that users see and interact with and the backend which contains web server, app logic [30]. The frontend sends requests to the backend as per the user interactions and receives the corresponding responses. The requests could be for a webpage or for a smaller component/feature within a webpage. Accordingly, the web server may respond back to frontend with a page HTML or a JSON object respectively. Further, not each user interaction may result in a request being sent to the web server, for instance opening up a dialog box or expanding drop down menu options. This is due to javascript code running in the frontend.

The request sent to server by frontend typically contains the a URL, method, headers and body and the response received contains the a response code, headers and body as shown in Figure 2.2. The URL stands for Universal Request Locator and is a reference to app resources. The method indicates the action to be taken on that resource. It can be GET(read), POST(create), PUT(modify) or DELETE(delete).

```
> url = new URL("http://www.website.com/some-object-path?param1=val1\&param2=val2")
URL {
  href: 'http://www.website.com/some-object-path?param1=val1&param2=val2',
  origin: 'http://www.website.com',
  protocol: 'http:',
  username: '',
  password: '',
  host: 'www.website.com',
  hostname: 'www.website.com',
  port: '',
  pathname: '/some-object-path',
  search: '?param1=val1&param2=val2',
  searchParams: URLSearchParams { 'param1' => 'val1', 'param2' => 'val2' },
  hash: ''
}
```

Figure 2.3: Structure of a typical request URL

The header contains useful metadata like cookies, content-type, language, encoding..etc. The request body is usually empty for GET requests. For other method requests it contains information like what values must the newly created object have or what should it be changed to. Similarly, the response body contains the page HTML or JSON data. It is usually compressed in gzip format for faster transmission.

Note that we define a resource as a request sent to the server that the server accepts. In other words, the server responds without a client side error or 400 HTTP response code. Additionally, we define an object as some data for which a state is maintained in the app database, for instance a repository in Gitlab or a paper submission in Hotcrp. Therefore, although resources or requests themselves are stateless, they can create or modify objects, affecting the app state. Typically, requests with method - POST create new objects and requests with method - GET retrieve objects/pages.

The request URL typically comprises of protocol, hostname, pathname and search/input parameter fields as described in Figure 2.3. The hostname is the address of the system where the server is running. Pathname is the directory path or an endpoint within the application. Search or input parameters reference specific objects of that endpoint.

The users interact with the frontend, also known as User Interface (UI), through a browser. These interactions can be automated via browser automation tools [47]. Such tools are usually used to perform end to end testing where we can automate user defined interactions to specific page's HTML elements and assert the expected response of the UI.

2.3 Evolutionary Algorithms

Evolutionary algorithms (EAs) are heuristic search algorithms used to find approximate solutions to computationally hard problems that cannot be solved in polynomial time. They leverage the Darwinian theory of evolution or survival of the fittest strategy as seen in biological evolution of species.

A solution or, in evolutionary terms, a chromosome of an individual, consists of smaller building blocks called genes. A set of genes are joined in a sequence to form a chromosome. The idea is that with evolution's natural selection, individuals with stronger chromosomes will persist in the population while weaker chromosomes will perish.

Genetic algorithms are popular EAs [51]. They are used for problems where it is easy to construct

some solution but hard to construct the optimal solution. GAs comprise of five main steps - initializing population, crossover/mutation, fitness/cost function and selection [45].

The GA starts by initializing a population of solutions usually randomly constructed. Next, crossover and mutation operations are performed over the existing population to produce new solutions in the population. All solutions are assigned a fitness value by the fitness function which defines the optimization criteria. Following that, the fitter solutions are selected to survive to the next generation and rest are discarded. The new generation follows the same steps - crossover/mutation, fitness value assigning, selection and this process is repeated until the termination condition is met. The termination condition is usually set to a fixed amount of run time, generations or if an acceptable fitness value is reached. In other words, the GA aims to find an optimal solution or 'sequence of genes' as per the fitness function, in a very large search space.

The population is generally initialized by randomly selecting genes and creating sequences. A higher diversity in the initial population usually leads to better results [45]. This is because with diversity, the algorithm is able to overcome local optima, allowing it to better explore the search space before converging on a solution.

There are many strategies for selection steps such as the tournament selection, rank selection, Truncation selection [45]. Selection strategies are selected as per the problem since no one strategy works best for all. The goal is usually to ensure faster convergence and diversity in population is maintained to avoid getting stuck in a local optimum.

The crossover step produces new sequences in the population from existing sequences. In crossover, two parent (p1,p2) sequences are broken into two parts - head(H), tail(T) and recombined to produce two new sequences. For instance - Hp1+Tp2, Hp2+Tp1. Similarly, many other strategies have been proposed [45]. Mutation also produces new sequences but it only needs one parent sequence. Some mutation strategies include - randomly swap two genes of the sequence, add/delete a gene. Furthermore, mutation can be performed based on some feedback to make the algorithm adaptive. For instance, when the algorithm is converging or the fitness values aren't improving over generations, the probability to mutate can be increased and similarly when there is high fluctuation/variance in fitness values, the mutation probability can be decreased.

Note that evolutionary algorithms are non deterministic. Multiple runs may lead to different results. Therefore, these algorithms are usually evaluated over multiple runs. Also, they are non terminating unless a terminating condition is specified like runtime or target fitness score.

2.4 Gestalt Pattern Matching

The Gestalt Pattern Matching is a method to measure the similarity between two strings. This approach is designed to find how similar two strings are, resembling human analysis. It uses the Longest Common Substring (LCS) to recursively keep matching longest contiguous characters and calculate the ratio as the number of matching characters divided by the total characters in the two strings [50]. Concretely,

$$SimilarityRatio = \frac{2 * M}{|S1| + |S2|} \tag{2.1}$$

where M are the number of matching characters between strings S1 and S2, $|S1|$ and $|S2|$ are number of characters in S1 and S2 respectively.

The time complexity of this algorithm is cubic polynomial time in the worst case. The is because finding LCS takes $O(|S1| * |S2|)$ in space and time. And this would be repeated $|S1|$ times in worst case. Thus, in total the algorithm takes $O(|S1|^2 * |S2|)$ time.

Chapter 3

Design

We designed AuthZee in a modular style where each module can run independently and communicate with its peer modules. There are four key modules - Simple crawler, Evolutionary crawler, Public Filter and Vulnerability Detector. The crawler modules discover new pages and record all requests originating from those pages. The Public Filter module filters out all the shared or publicly accessible URLs, so only private URLs are sent to the Vulnerability Detector. And finally, the Vulnerability Detector module automatically checks the authorization of a URL.

The simple crawler generates a stream of new URLs discovered by crawling the target web app. This crawler is fast at discovering static links in a webpage however, it cannot explore the user generated content that requires complex user interactions to be discovered. This is because it cannot perform dynamic multi step interactions like creating an object, submitting a form, selecting from a drop down menu, updating object settings, handling javascript dialog boxes. The evolutionary crawler attempts to solve this problem. It leverages an evolutionary algorithm to perform multi step interactions on a webpage like a human user. However, the evolutionary crawler is much slower. Therefore, this crawler prioritizes which page to crawl first based on the *page interactivity score*. The page score is calculated by counting the number of interactive elements contained in the page for instance input, button, textarea etc. This is done to avoid spending time on non-interactive pages which are usually public pages. Once a new object is created by an evolutionary crawler, we need the simple crawler to quickly crawl any new static links associated with that object, which in turn will be shared with evolutionary crawler for further object creation. Thus, using simple crawler in combination with evolutionary crawler enables a deeper exploration of the web app.

Concurrently, all the pages discovered or requests made to the app by the two crawlers are sent to the public filter module. Here, the requests that are public or shared among multiple users are identified and discarded, and the remaining ones are sent to the vulnerability detector. The public pages must be filtered out because they confuse the vulnerability detector causing it to produce false positives. And the requests that fail authorization checks are reported by the detector.

The following subsections will explain the modules in more detail.

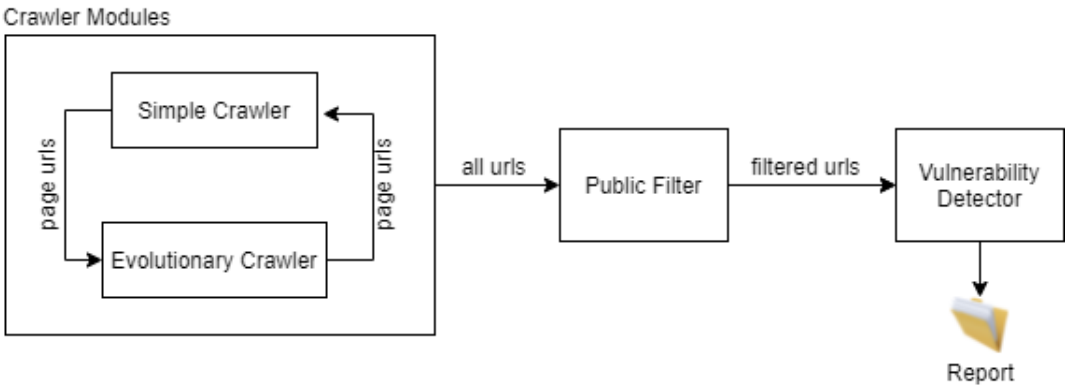

Figure 3.1: Block Diagram of AuthZee

3.1 Simple Crawler

The simple crawler crawls the target web application in a bread first search (BFS) fashion. It takes in a target URL as input and then visits all the links on the target URL's page. This generates new URLs and for each new URL, all links on those pages are visited and this process continues until all the pages associated with the target domain have been visited. This way of discovering URLs is called website crawling and it isn't novel.

Existing crawling techniques do not perform well over web applications of our interest that require user authentication to explore app features and resources, assuming authentication cannot be bypassed. This is because the app resources are hidden from public/non registered or unauthenticated users and crawlers by themselves do not contain user credentials or authentication capabilities. Therefore, we designed the simple crawler to accept as input - the user credentials and instructions to login. For instance, navigate to the login URL, fill specific HTML elements like username and password input fields and click the login button. The HTML elements are located using CSS identifiers like id, class name or other HTML tag attributes. The tester must pass this login info to the crawler. This allows the simple crawler to automatically crawl as an authenticated user and explore more deeper resources. Note that the simple crawler does not automatically register or create a new user account in the application. We manually create user accounts and then pass the login instructions as input to our crawler.

The simple crawler also captures dynamic component level URLs. Web applications featuring interactive user interfaces are mostly Single Page Applications (SPAs) or make heavy use of AJAX (Asynchronous Javascript and XML) [21] [2]. SPAs are a style of development where the page is made up of multiple smaller components and instead of loading the entire page for any request (A request can be sent to the server as a result of user interaction with a page component) only that specific component gets reloaded. These kinds of applications leverage AJAX to communicate with servers without reloading the entire page [2]. In other words, visiting a page can be accompanied with other AJAX (XMLHttpRequest) requests to fetch the individual components of that page. The simple crawler captures such AJAX requests during its crawling. These requests are of interest because even if a page request is access controlled, their individual components or function requests may not have proper access control.

Finally, the crawler also performs some validations like dropping out of domain URLs and keeping

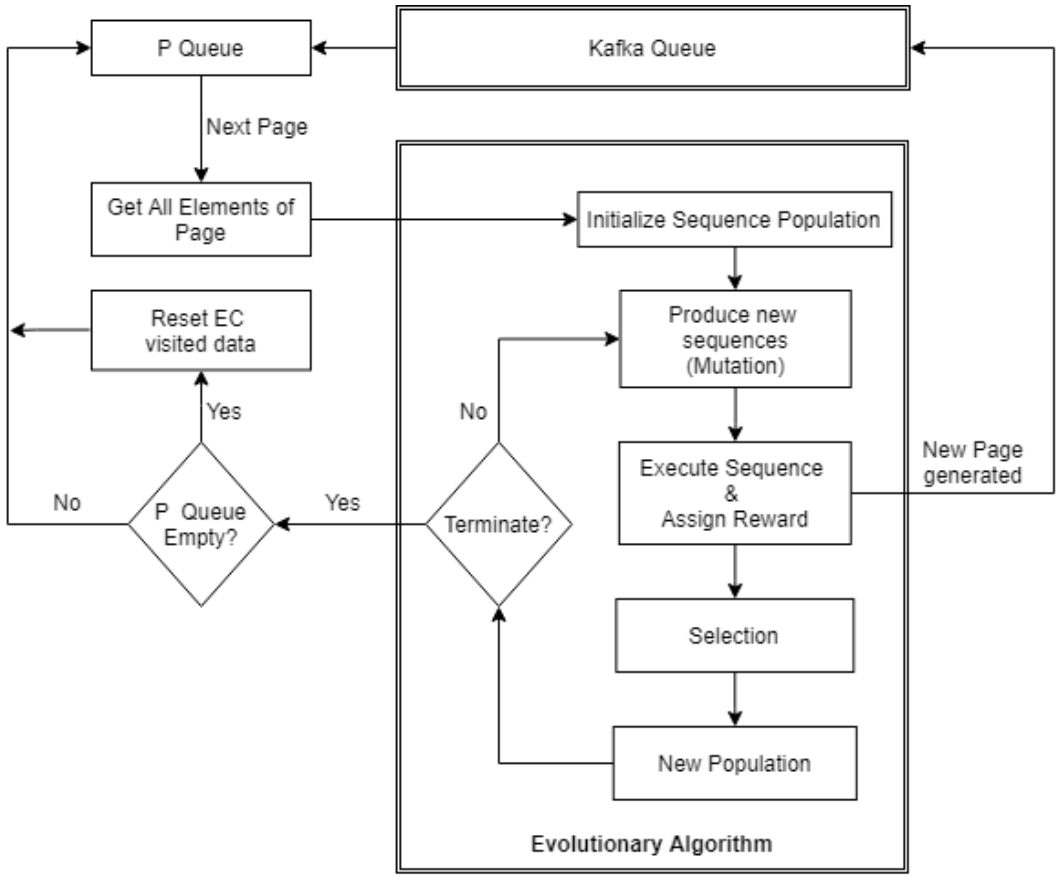

Figure 3.2: Block Diagram of Evolutionary Crawler

track of visited URLs to avoid redundant loops.

3.2 Evolutionary Crawler

In web security, crawlers are primarily used in web application vulnerability scanners such as Burp spider [22] and ZAP active scan [14]. These tools can automatically crawl through the target website and detect vulnerabilities. However, they can only find certain types of vulnerabilities. Authorization vulnerabilities like IDORs resulting from broken access control cannot be detected by web vulnerability scanners due to their inability to create new objects and detect unauthorized accesses from server response [14]. Scanners perform poorly for applications with a high number of dynamically generated pages where new pages or objects are generated because of user actions. Furthermore, many of them do not handle dynamic AJAX content and complicated Javascript interactions, for instance new components like popups, dropdowns, modals etc becoming visible as a result of some user activity like typing and clicking [33]. Such interactions are important for extensively exploring app features and discovering new resources like URLs or API endpoints which the simple crawler cannot discover. Thus, there exists a gap between discovery of app resources by automated simple crawling and manual user interactions.

Most applications, depending on their design and logic, have a multi step process (in terms of user interactions) to create, modify or delete objects, thus changing the user account state or app state.

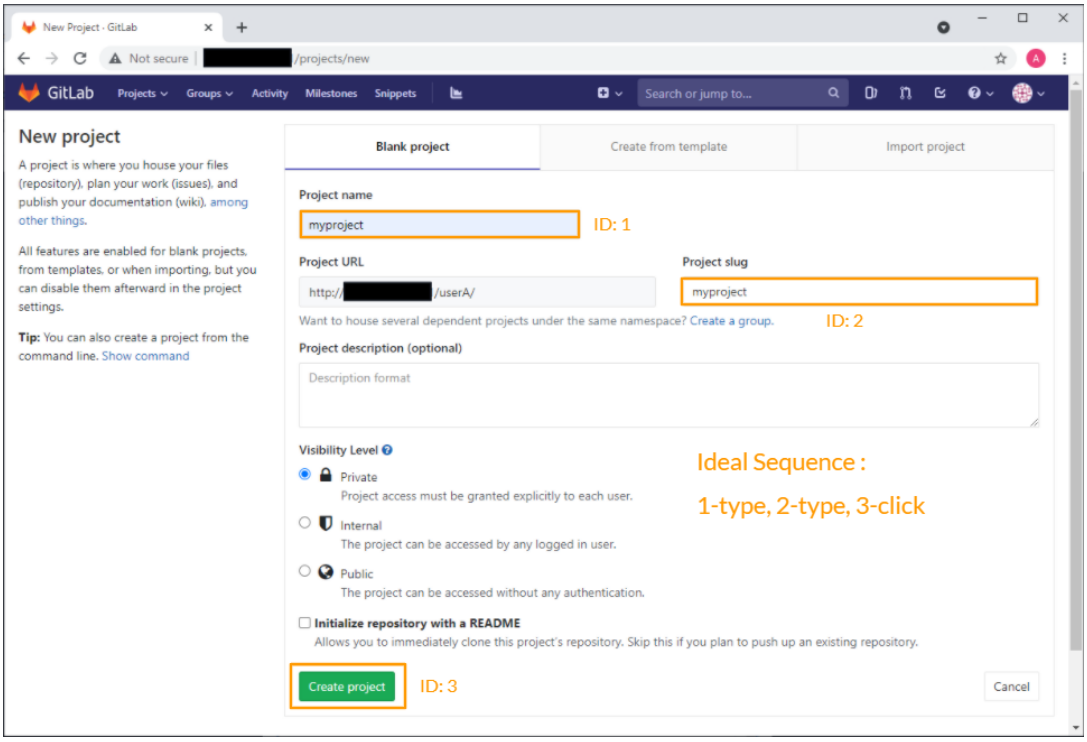

Figure 3.3: An ideal element-action sequence on the '/projects/new' page in Gitlab

However, for a crawler that only looks at the page elements like hyperlink, text input, button etc from the page HTML, it is very challenging to determine what exact order of interaction will result in a change in app/useraccount state. The Burp crawler does try to look for form elements on web pages and submits them; however not all objects are created or updated by submitting forms. Also, submitting forms generally results in reloading the page. Therefore, most AJAX communication that does not accompany page reloads cannot be captured after submitting forms. Furthermore, there may be multi step user interactions required before the form element is even visible on the page.

To close the gap mentioned earlier, we need to find the ideal sequences of user actions that yield either a new component visible on the web page or a new request sent to the server. Figure 3.3 shows an ideal sequence which would yield a new request for creating a new project in Gitlab. This is quite challenging as there can be a large number of possible permutations of user actions with elements for a web page, especially when the length of sequence is also unknown. A naive randomized brute force search is therefore impractical. Our evolutionary crawler tries to close this gap by leveraging an evolutionary genetic algorithm. The genetic algorithm attempts to search specific sequences of user interactions when simulated on the webpage, result in the web app sending new requests to the web server, thereby performing a higher degree of crawling. Thus, it attempts to find the sequences of interactions that a human would do to interact with a webpage.

Genetic algorithms algorithms apply the concept of natural selection which states that given an environment, only the fit or most adaptable survive over many generations of evolution. Here, we set the environment and a reward system such that over generations, ideal sequences have a higher chance of survival. Thus, finding ideal sequences that produce new requests when executed on the webpage.

We apply the genetic algorithm to our problem as follows. A gene is defined as an element-interaction pair. For instance, an <input> HTML element can be interacted with by clicking, typing, hovering etc. And so the resulting genes for that input element could be input-click, input-type, input-hover etc. Thus, our search space is then made up of different sequences of such genes. The goal of the algorithm is to find certain sequences which when executed on the webpage will generate new requests that Create, Read, Update or Delete application data objects, also known as CRUD requests.

Initially, all the interactable elements are detected from the page html and they are combined with different user interactions to create genes. The algorithm starts with an initial population of randomly creating sequences of genes. A random sequence is created by picking a set number of elements equal to the initial sequence size, from the webpage in random order and assigning them an interaction. At each iteration or generation, each sequence in the population is evaluated by the heuristic based reward system which assigns a score to it. The sequences are then sorted based on their score and an *elitist selection strategy* - always picking top X number of sequences, is used to choose the sequences that will propagate to the next generation. The top scoring half of the population propagates to next generation and the rest gets discarded. Now, in the next generation, we produce new sequences from the top scoring population of the previous generation. The basic idea of evolutionary algorithms, that is - higher scoring parents will produce even higher scoring offspring perfectly applies to our use case. More concretely, a sequence with strong genes will have a high score and thus those genes will be passed on to next generations. Note that high scoring sequences are the sequences that have a higher chance of changing the app state.

The following paragraphs will explain how new sequences are generated from previous generations and how the reward system assigns a score to a sequence.

3.2.1 Generating New Sequences

When producing new sequences, we care about the order in which the element-interaction genes are executed due to the underlying dependency constraints. For instance, interacting with one element may make the next element in the sequence visible on the web page. The popular crossover and mutation techniques may often break the dependency order and thus mostly produce lower scoring sequences than previous generations. To avoid this, we add web page specific heuristics. We detect if interacting with an element makes new elements visible and mutate the sequence by inserting one of those dependent genes obtained by assigning an interaction to the newly visible element after the element in the sequence that was interacted upon previously. Also, at each iteration, some sequences are still generated randomly as a source of new genes.

The mutation operation increases the size of the sequence as we are inserting new genes. We implement two things to ensure the sequence size does not increase without bound. Firstly, we make sure the sequence has no duplicate genes. Second, we assign a gene score to individual genes. The gene score indicates the desirability of a gene. And the genes with low desirability are removed from all sequences. The reward system assigns the gene score and sequence score.

We emphasize that the idea that parents with higher fitness produce fitter offspring applies here. Since we reward sequences based on browser feedback as listed in Table 3.1, a parent with higher score

Browser UI behavior after element interaction	**Reward**
Sample file uploaded on UI	+ve
Typing successful on a page element	++ve
New elements visible	++ve
New request sent	+++ve
Chosen valid option from a select element	++ve
Outside domain request sent	—ve
Duplicate request sent	-ve
No UI change / timeout	–ve
Element invisible	–ve
Browser disconnect/restart	—ve

Table 3.1: Reward/Penalty for specific UI behaviors after interacting with an html element

indicates that it is more closer to an ideal sequence. Therefore, mutating the parent with a higher score has a higher chance of turning into one of the ideal sequences, compared to mutating a lower scoring sequence. Hence the sequence population improves over generations eventually leading to ideal sequences that yield new requests when executed on the webpage.

3.2.2 Reward System

The reward system guides the evolution towards a population of sequences that produce desired outcomes like new requests generated or changes in app user interface (UI). This is also referred to as the fitness or cost function in traditional genetic algorithms. Here, the fitness or reward is calculated dynamically upon executing the particular sequence or gene, meaning we perform the interactions on elements in the web page as per the sequence and calculate the reward based on feedback from the app UI.

There are two kinds of scores assigned here - sequence score and gene score.

The sequence score determines how promising the sequence is in updating account state. Since the actual request generating sequences are very sparse compared to the entire search space of sequences, we need to incorporate reward shaping in order for the evolution to be efficient and practically useful. Therefore, instead of only rewarding the sequences that generate new requests, we reward sequences based on interactions that may lead towards request generation. For instance, if an input field was successfully filled in, if a file was successfully uploaded, new elements became visible, or when a new request was seen. For the latter, we also assign a bigger reward. Table 3.1 lists the rewards for different UI behaviors after interaction with an element. In other words, we reward the sequence when it is realized that the interaction with an element was successful or the gene was successfully executed. Similarly, the interactions that are impossible or unfavourable are penalized. For example, when an out of domain / previously seen request is generated or if the sequence took too long to execute, or the page element of the gene is not yet visible since the visibility could be dependent on interactions with other elements. Over generations, this promotes interactive sequences. However, when there are many unproductive genes in the search space, it may take many generations to evolve to an interactive sequence of genes.

Therefore, we introduce gene score to reduce the search space by getting rid of unproductive genes.

The gene score tells us how useful that gene is by itself. Initially, a set score is assigned to all genes. Over generations, the gene score is affected by two factors - Intractability and Novelty. For intractability, a reward or penalty is awarded depending upon whether the specific interaction for that element was successful. For example, performing typing on hyperlink or clicking on text fields, unsuccessfully typing into a disabled input and clicking an element that has no effect on the web page. This criteria is similar to the one for assigning sequence score in that it gets response feedback from the web app's UI but it is more focused on the individual gene itself. We do this to avoid flooding the population with genes that do not make meaningful interactions and make evolutionary search less efficient. The purpose of gene score is only to eliminate low scoring genes from the population of gene sequences. It is not used during the evolution selection process like the sequence score is.

Note that genetic algorithms have been traditionally used for optimization problems, where the goal is to optimize the fitness/cost of a population of solutions through the process of natural selection. The optimization usually converges, meaning that as time passes, the fitness/cost of the solution will improve less and less until it doesn't improve at all, indicating that the algorithm has converged and the resulting solution is close to optimal if not optimal. However, this does not quite fit with our needs. Instead of finding one optimal solution, we want to find as many solutions as possible. Here, a solution is any sequence of element-useraction pairs (aka genes). Ideally, the more permutations of element-action pairs we explore, the higher the chance of finding a sequence that generates a request to create/modify/delete objects (POST, PUT, DELETE). Therefore, we take a novelty search approach, to guide evolution to yield many useful sequences.

To this end, the gene score is also affected by the novelty of the gene in the population. It gets reduced at the end of each generation by a set amount. Thus, a lower score also means that the gene has been in the population for a long time. Since we don't want the algorithm to converge, as is the natural tendency of genetic algorithms, we remove the genes that are very old so new genes have a higher chance to be explored, even if they initially don't look promising.

3.3 Public Filter

For real world apps, there can be many pages and even dynamically user generated resources that are supposed to be accessible by multiple users as per the app logic. This can cause a large number of false positives when detecting authorization vulnerabilities. It is because the detector relies on differences in responses of users to verify authorization. If multiple users receive the same legitimate response, the detector wrongly reports this as an authorization vulnerability. Therefore, the purpose of this module is to mitigate this issue by inferring developer intention.

We try to identify if a resource is intended by the app developers to be public or shared as follows. Our assumption is that if a user is able to discover a URL or resource by navigating on the website without forced browsing or protocol field substitution, then the developers intended for that resource to be accessed by that user. Thus, we run two instances of the simple crawler, but they crawl logged in as two different users, let's say userA and userB. We call the second simple crawler instance the twin

crawler. So if a resource is discovered by both simple crawler and twin crawler i.e both userA and userB can access that resource then it is a shared or public resource and must not be tested for authorization failures. Then we filter out such public resources i.e the requests or pages that were discovered by both the simple and twin crawler and only send the supposedly private resources to the vulnerability detector module. Thus, mitigating false positives during detection as shown in section 5.3.2.

3.4 Vulnerability Detector

This module is responsible for automatically detecting if a request to a resource fails authorization in an application agnostic way. Previous solutions have either relied on error codes in the response or manual visual analysis by human eyes. As we mentioned in the introduction section, the error codes are unreliable as the app may not always return proper error codes, and instead return 200 OK (success) for every request, thus resulting in false positives when checking authorization.

The vulnerability detector resends each website request captured earlier by the crawlers, three times with three different user's session cookies. We refer to this as a triad run. The session cookies are obtained after automatically logging in with the user's credentials. Therefore, this module requires three sets of user credentials. The first - userA that was used during simple and evolutionary crawler, the second - userB used during twin crawler and the third - a userC that must be at the same privilege level as userB. So, ideally with a successful authorization implementation, we expect the request sent with userA's cookies to contain a legitimate authorized response or the ground truth. And since, the request made is for userA's private resources, the response received with userB and userC's session cookies must be different. Otherwise, there exists an authorization failure.

Note that it is not enough to run the request only twice, for example with userA and userB's session cookies. Although by comparing the page html of the two responses, we will still be able to determine if response to userB is different from response to userA, however we would not know if that is because the page content is different or just because of user specific data such as username, user-email on the webpage. For example, when we login to an account, it is fair to assume that for most apps, we will see some indication of who is logged in like our username at the top-left corner of the page. This static user specific content can result in false negatives or missed authorization failures since they would always make the responses different. Therefore, we must find a way to isolate the user specific content, so we ignore it when comparing with the ground truth or the response to userA. This is where userC is needed. Since, both userB and userC are at the same privilege level and neither must have authorization to access the resource, their responses must be similar with the only difference in the user specific content. Thus, we can detect the user specific data and ignore it when comparing userB's response with the ground truth.

To make comparison easier, we work with similarity ratios. We calculate the similarity ratio between userA's HTML response and userB's HTML response - simRatioAB and between userB's response and userC's HTML response - simRatioBC. The similarity ratio is calculated using gestalt pattern matching by Ratcliff and Obershelp [50] as explained in section 2.4. Although it is a cubic polynomial time algorithm in the worst case, in practice this is not the bottle neck of the design because web server responses cannot be infinitely large as this would affect the app quality

For our use case, the simRatioBC tells us what percentage of content is common between userB's response and userC's response. Or (1- simRatioBC) is the percentage of content that is different or user specific. Thus, the % difference between responses of userA and userB must be greater than the % user specific content or (1-simRatioBC). Thus,

$$(1 - simRatioAB) > (1 - simRatioBC) \tag{3.1}$$

when authorization is working correctly. This is same as simRatioAB < simRatioBC. Concretely, there exists an authorization vulnerability for a resource if

$$(simRatioBC - simRatioAB) \leq 0 \tag{3.2}$$

However, this is too strict and we need to allow for some degree of margin. Therefore, we introduce a margin factor (M.F) which can be set to a small value ex. 0.001. In conclusion,

$$simRatioBC \ - simRatioAB \ \leq \ MF \tag{3.3}$$

indicates other users were able to access userA's resource, implying there is an authorization vulnerability. Thus, we are checking if the response to userB was more similar to the ground truth(userA's response) than userC's response.

Chapter 4

Implementation

As mentioned earlier, we designed a modular system to automatically detect authorization vulnerabilities in web applications and this section provides more details on the frameworks and packages used to build it.

4.1 Custom Testcafe Framework

The simple, twin and evolutionary crawlers are built using a customized version of testcafe 1.13.0 [7]. Testcafe is an end to end web application testing framework. It provides browser automation capabilities along with useful features like capturing all requests-responses sent and received, user roles to automatically login as a user, checking if an html element is visible on screen, interacting with an element, running javascript code in the browser, automatic waiting for elements to appear [7]. This allows us to identify desired behaviors like if new elements were visible on the screen after a click interaction, if a typing interaction was successful on an element, if the html element - 'select' was successfully clicked to select one of its options, and if a new request was generated due to some sequence of interactions. This is used by the reward system where sequences showing desired behaviors are rewarded and similarly, uneventful sequences are penalized. Since these behaviors are application agnostic, we can hard code the reward system to look for such behaviors and still be able to generalize the crawling for all web apps. Furthermore, testcafe also supports monitoring the automated user interactions in real time, taking snapshots or recording video of the activity on the browser [15].

However, testcafe expects the tester to know the app structure where the tester must specify the HTML elements to interact with by providing element identifiers like CSS class name or id. This is not ideal for crawlers since you don't know the element ids beforehand. Furthermore, Testcafe is designed to halt on encountering a failure or unsuccessful interaction and for a crawler that is an undesirable quality, especially for evolutionary crawler where we often encounter unusual interactions. Therefore, we modify the testcafe package to keep running even after errors. And on discovering a new page, we fetch all interactable elements from the response HTML dynamically during run time and assign them internal ids to reference them during evolution. We also lower some wait timeout values to speed up the crawling. Testcafe is configured to use chrome browser in AuthZee but any other browser can be used. Ideally, a browser that is most compatible with the target app UI should be selected.

Futhermore, we store the data items describing the current crawling state, such as url queue, visited urls, current population of sequences..etc. This allows us to handle any unexpected interruptions in the crawling process like loosing browser connection or opening a plain text page which causes testcafe to hang, in which case it automatically restarts after browser heartbeat timeout. And as the crawler restarts, it picks up from where it left off, thus avoiding loosing previous progress. This however does lower the performance since we now must write to the hard disk during each crawling iteration. This also requires enough memory for the metadata like current population, gene scores, element dependencies, visited pages, queue and running Kafka. The memory usage thus depends on the evolutionary crawler parameters and target app.

All modules are built in Javascript mainly because Testcafe only supports Javascript. Furthermore, we use the difflib package to calculate similarity ratios in the triad detector module. And to run the experiments, we leveraged the pm2 process manager package for nodejs.

4.2 Apache Kafka

Apache Kafka is used to transfer data between modules. Kafka is an event streaming framework that provides permanent storage [3]. AuthZee uses it as a persistent queue. The simple crawler and evolutionary crawlers send new discovered page URLs to *pageurls* queue. They also consume from the *pageurls* queue but in different consumer groups. This enables simple crawling and evolutionary crawling to consume the queue separately as though there were two separate queues. The two crawlers also send all URLs to *allurls* queue which are consumed by the filter module. Concurrently, the twin crawler sends all URLs to *twinallurls*. The filter module consumes from *allurls* and *twinallurls* to filter out public URLs and sends the private URLs to *filteredurls* queue which is then consumed by the triad detector.

4.3 Bypassing Browser Security

Browsers implement a security feature that prevents users from accessing secure cookies using javascript. App servers can mark cookies as secure by including an attribute flag - "HttpOnly=1" at the end of the cookie [54]. This instructs the web browsers that this cookie must be accessed only through HTTP, thus preventing it from being disclosed through javascript's 'document.cookie' call. All modern applications use this feature to protect from cookie theft against attacks like Cross Site Scripting (XSS), Cross Site Request Forgery (CSRF).

This security feature however prevents AuthZee from receiving the user session cookies of userB and userC. These cookies are needed to perform triad tests which replace userA session cookies with userB's, in the request logs discovered during crawling. And then this request log is resent to the app server. Thus, it enables sending the exact same request that userA sent but with an unauthorized user's session cookies - ex. userB. The same is repeated for userC.

Since the secure session cookies cannot be fetched from the browser, testcafe's ability to auto login and log requests,responses is leveraged to procure the secure cookies. All applications upon successful login, redirect users to their account dashboard or home page. Since this page would contain the user's private data, the request sent to fetch this page must contain the user session cookies. Therefore, the

user session cookies are successfully extracted from the last request logged following a user login.

Bypassing the HttpOnly flag allows AuthZee to perform triad tests for AJAX requests as well as POST/PUT requests. However, the requests returning empty responses like empty string, array, json object - "", [ ], { } are not triad tested, since it would lead to a false positive.

Further note that because we use cookie swapping method to find IDORs, we can test all URLs for unauthorized access unlike the parameter tampering method where the URL must contain objectIDs, which are then altered.

4.4 Additional Requirements

This subsection states the initial setup needed to use AuthZee. It also clarifies additional steps that our implementation takes for optimal results.

The vulnerability detector requires three sets of pre-created user credentials and css selectors of the login form to automatically login to the app. This one time small effort to manually register 3 users is required to run AuthZee on a web app.

Further, the genetic algorithm in evolutionary crawler, for best results, requires tuning of some parameters like the sequence length, population size, number of generations or iterations and behavior rewards and penalties to evaluate the sequence and gene score per iteration. In the future, this can be extended to be adaptive where the parameters are not fixed and adapt to the environment based on some feedback. For instance, if the sequence scores are converging over iterations then increase the novelty reward. Similarly, the triad detector may also require tuning of the margin factor (M.F) parameter. A larger MF can result in increased false positives but setting it too small could miss some of the authorization failures.

Our design takes a long time to complete. This is mainly because the evolutionary crawler runs many iterations per page, hence the bottleneck in our design. However, this is not a limitation as we can improve crawling throughput by horizontally scaling with additional evolutionary crawler programs that consume from the same kafka queue. Kafka allows programs running on different machines to consume it's queue. This will allow us to concurrently run multiple evolutionary crawler programs that distribute the crawling load and thus reduce the total amount of time it takes to fully explore the app.

Another point worth mentioning is that during crawling we need to ignore the elements that logout or sign out from the user account. These elements exit the user account space and cause the crawler to explore the public space. This isn't a major problem as those public pages will be filtered out later, however it does waste time. Therefore for optimal performance, the tester needs to specify which element to ignore by providing it's html/css identifier. In our implementation, this is hard coded by checking for specific keywords like 'logout' or 'sign out' before clicking an element.

Chapter 5

Evaluation

We formulated the automatic blackbox authorization vulnerability detection problem by putting together two subproblems - resource discovery and authorization detection. Therefore, we evaluate our implementation over the two metrics: number of resources discovered and number of false positives during detection. The following subsections explain the experiment setup and different experiment runs to evaluate AuthZee.

5.1 Experiment Setup

We chose 7 open source web applications as our targets to evaluate our apps - Openstack [8], Gitlab (Community Edition) [11], HotCRP [9], Overleaf (ShareLatex) [10], DokuWiki [5], HumHub [13] and Kanboard [12]. Openstack is an infrastructure-as-a-service application and is used to deploy and manage cloud infrastructure. Gitlab is a code sharing application with many features such as git repository, wiki, issue-tracking and continuous integration. Hotcrp app is used to manage academic paper review processes. Overleaf app is used to share and collaborate on papers in latex. Dokuwiki is a wiki website. Humhub is a social networking application. And finally, Kanboard is a project management application that uses the Kanban methodology. These web apps were purposely chosen as they support user account features where users can create and collaborate on in app resources, thus enforcing access control policies that can be checked for vulnerabilities.

Our resource discovery experiments consisted of running the simple crawler with BFS crawling strategy and evolutionary crawlers with different evolutionary parameters (ex. sequence size *'s'*, population size *'p'* and number of iterations per page *'i'*) over all target web apps. We ran the evolutionary crawler multiple times with different combinations of s, p, i as shown in table. The crawling captured and saved following data in persistent storage - page URLs, all URLs including AJAX requests, filtered URLs, requests with a non-GET HTTP method. The latter signifies how many different object creator/modifier requests the crawler discovered. From the results, the most suitable combination across all apps was picked for further experiments.

The detection experiments used data generated by the crawlers to run the vulnerability detector for each URL resource. The resources failing authorization as per the triad test were reported and the false positives were manually identified by human testers. Originally, the margin factor (MF) was set to 0.001

Each experiment was run for 24 hours on a different but identical instance of the target webapps using their open source installation guides. We hosted all docker containers of the webapps on our research group's virtual machines running on Ubuntu 18.04 OS.

Further, each experiment was evaluated by four measurements - pageurl count, allurl count, filtered url count and number of non-GET requests. The pageurl count is the number of webpages that were visited during the experiment. Allurl count tells us the total number of requests sent by the frontend to backend including the requests for webpages. We configure testcafe to only log the requests that accept content type - 'text/html', 'application/json', 'text/plain' from the server, since other requests for fonts,icons,css do not contain user sensitive data. Concretely, pageurls are requests that only accept 'text/html' while allurls are requests that accept all three types of content. We filter all urls using the public filter module to generate filtered urls. And finally, the non-GET requests are any request with a method different from GET, for instance POST, PUT, DELETE. We count the non-GET requests as these are primarily responsible for a change in app state. Thus this is a measure of change in app state.

5.2 Discovery of Request URLs

In this section, we perform multiple experiments with the target app to evaluate our crawlers and their URL discovering ability

5.2.1 Evolutionary parameter tuning

As mentioned in the setup, we tune the evolutionary parameters - sequence size *'s'*, population size *'p'*, iteration amount *'i'* by running the evolutionary crawler multiple times, varying these parameters. s indicates the initial number of element-interaction genes per sequence that algorithm starts with. This doesn't significantly affect the evolution results because the sequence size is not fixed and mutates with iterations. Next, p indicates how many sequences we execute in one iteration. i indicates the number of generations spent per page searching for new requests. We run nine evolutionary experiments for each app, where each experiment has a different combination of s,p,i parameters. To understand the influence of a single parameter in the evolution, one parameter was tested with three different values while the other two were kept unchanged. Thus, three parameters result in nine combinations, hence nine experiments. Table 5.1 to 5.7 show the results of these nine experiments for all our target apps.

From the results, there is no clear winner across all the target apps. This can be attributed to two factors - app UI design and evolutionary crawler's non deterministic behavior. For apps like Gitlab and Hotcrp with many interactive features in their UI, we notice that a higher iteration value gives better results with higher number of non-GET requests. Similarly for apps with simpler frontends like Dokuwiki and Humhub, lower input parameters generally give better results. This is because of more pages being fully iterated over.

We note that our evolutionary crawler is non deterministic due the use of random initialization and mutation. Therefore, multiple evolutionary runs with the same input parameters may give different results. We showcase this by running evolutionary crawler with same parameters s=5, p=10, i=25 three times as shown in Figure 5.1. The results, although different each time, have the same order of magnitude.

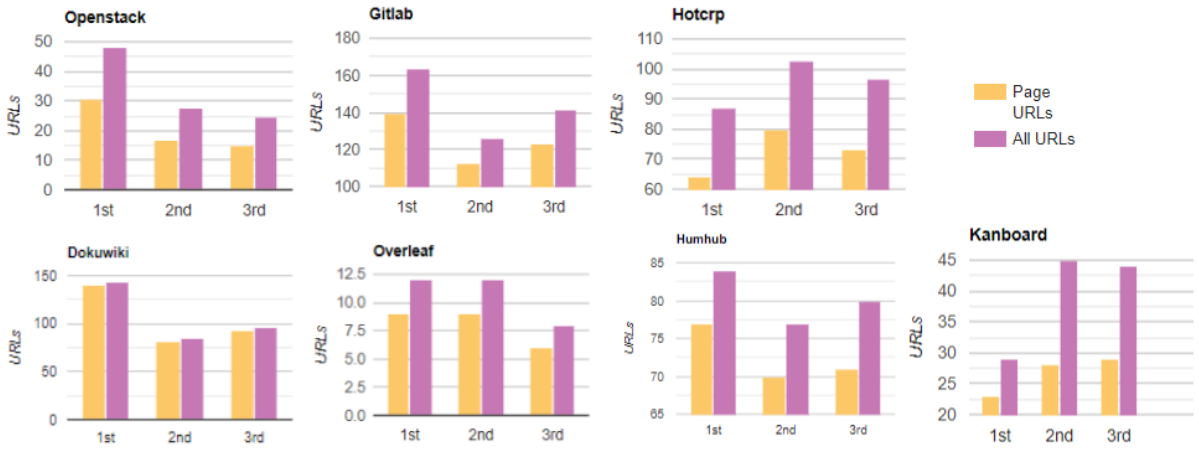

Figure 5.1: Illustrating non-deterministic behavior of Evolutionary crawler. Results from three experimental runs with same parameters $s = 5$, $p = 10$, $i = 25$

We also measure the number of pages that were fully exploited, meaning all i evolutionary iterations were completed on those pages. This allows us to compare exploration with exploitation. Increasing input parameters s, p amd i allows us to increase time spent interacting per webpage, thus focusing more on exploitation. This inturn causes fewer webpages exploited since each experiment is run for 24 hours, meaning decreased exploration. In other words, increasing exploitation would take longer time exploring the same number of webpages. Thus there is a tradeoff between exploration and exploitation.

Furthermore, to compare experiments with different input parameters, we introduce an efficiency measure that we define as the number of all requests produced per unit interaction with the frontend. As a reference, the efficiency of the simple crawler is generally close to one, since each interaction with UI (clicking hyperlinks) generally yeilds a request. The results show that increasing input parameters typically decreased efficiency. However, this does not mean that lower input parameters are better since frontends rich in multi step interactions, would require larger values for p and i to successfully find a multi step interaction sequence that generates an app state changing non-GET request.

From the results, we picked the combination s=5, p=10, i=25 for future experiment as it gave consistently good results in all apps.

Seq size	**2**	**3**	**5**	5	5	5	5	5	5
Population size	10	10	10	**2**	**10**	**20**	10	10	10
Iterations	25	25	25	25	25	25	**10**	**25**	**50**
Results:									
page urls	12	16	31	31	17	16	28	15	16
all urls	19	36	48	53	28	28	44	25	26
filtered (private) urls	13	12	40	28	18	15	31	16	13
not-GET requests	4	5	4	3	5	4	6	5	3
Total UI interactions	6000	12000	38750	7750	21250	40000	14000	18750	40000
Efficiency	0.0032	0.003	0.0012	0.0068	0.0013	0.0007	0.0031	0.0013	0.0007

Table 5.1: EV Crawler parameter tuning for Openstack

Seq size	**2**	**3**	**5**	5	5	5	5	5	5
Population size	10	10	10	**2**	**10**	**20**	10	10	10
Iterations	25	25	25	25	25	25	**10**	**25**	**50**
Results:									
page urls	122	139	140	133	113	133	189	123	112
all urls	135	151	164	146	126	145	209	141	145
filtered (private) urls	112	128	146	122	97	120	105	120	120
not-GET requests	0	0	1	0	0	0	0	1	3
Total UI interactions	17000	15000	25000	9000	17500	40000	8000	21250	15000
Efficiency	0.0079	0.0101	0.0066	0.0162	0.0072	0.0036	0.0261	0.0066	0.0097

Table 5.2: EV Crawler parameter tuning for Gitlab

Seq size	**2**	**3**	**5**	5	5	5	5	5	5
Population size	10	10	10	**2**	**10**	**20**	10	10	10
Iterations	25	25	25	25	25	25	**10**	**25**	**50**
Results:									
page urls	78	78	64	58	80	64	135	73	110
all urls	103	103	87	82	103	90	160	97	141
filtered (private) urls	24	26	17	13	21	22	46	21	41
not-GET requests	1	3	2	1	2	8	2	3	20
Total UI interactions	16000	17250	35000	8000	40000	57500	42000	37500	95000
Efficiency	0.0064	0.006	0.0025	0.0103	0.0026	0.0016	0.0038	0.0026	0.0015

Table 5.3: EV Crawler parameter tuning for Hotcrp

Seq size	**2**	**3**	**5**	5	5	5	5	5	5
Population size	10	10	10	**2**	**10**	**20**	10	10	10
Iterations	25	25	25	25	25	25	**10**	**25**	**50**
Results:									
page urls	8	8	9	9	9	10	7	6	6
all urls	11	11	12	14	12	14	9	8	9
filtered (private) urls	6	6	7	10	8	10	4	3	4
not-GET requests	1	2	3	6	3	6	1	0	1
Total UI interactions	4000	6000	11250	2250	11250	25000	3500	7500	15000
Efficiency	0.0028	0.0018	0.0011	0.0062	0.0011	0.0006	0.0026	0.0011	0.0006

Table 5.4: EV Crawler parameter tuning for Overleaf

Seq size	**2**	**3**	**5**	5	5	5	5	5	5
Population size	10	10	10	**2**	**10**	**20**	10	10	10
Iterations	25	25	25	25	25	25	**10**	**25**	**50**
Results:									
page urls	60	79	141	87	81	25	58	93	131
all urls	35	40	63	27	26	15	23	27	48
filtered (private) urls	25	13	31	9	8	6	4	9	18
not-GET requests	6	4	6	5	3	2	2	3	6
Total UI interactions	19500	9750	27500	8250	15000	12500	11000	23750	65000
Efficiency	0.0018	0.0041	0.0023	0.0033	0.0017	0.0012	0.0021	0.0011	0.0007

Table 5.5: EV Crawler parameter tuning for Dokuwiki

Seq size	**2**	**3**	**5**	5	5	5	5	5	5
Population size	10	10	10	**2**	**10**	**20**	10	10	10
Iterations	25	25	25	25	25	25	**10**	**25**	**50**
Results:									
page urls	95	81	77	78	70	53	75	71	60
all urls	102	88	84	85	77	63	85	80	66
filtered (private) urls	48	39	39	32	33	23	37	32	19
not-GET requests	24	14	14	21	17	13	20	19	14
Total UI interactions	46500	32250	38750	18000	33750	45000	33000	31250	27500
Efficiency	0.0022	0.0027	0.0022	0.0047	0.0023	0.0014	0.0026	0.0026	0.0024

Table 5.6: EV Crawler parameter tuning for Humhub

Seq size	**2**	**3**	**5**	5	5	5	5	5	5
Population size	10	10	10	**2**	**10**	**20**	10	10	10
Iterations	25	25	25	25	25	25	**10**	**25**	**50**
Results:									
page urls	21	26	23	25	28	25	23	29	26
all urls	22	38	29	53	45	35	26	44	34
filtered (private) urls	21	37	28	52	44	34	25	43	33
not-GET requests	4	7	11	30	7	13	4	9	8
Total UI interactions	4500	17250	15000	6250	35000	62500	11500	36250	47500
Efficiency	0.0049	0.0022	0.0019	0.0085	0.0013	0.0006	0.0023	0.0012	0.0007

Table 5.7: EV Crawler parameter tuning for Kanboard

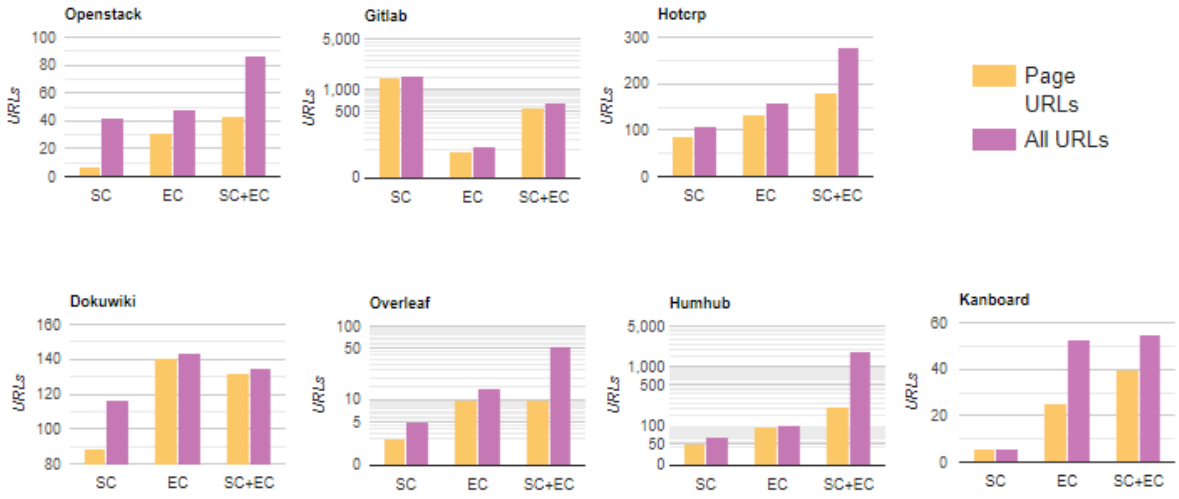

Figure 5.2: Comparing results of simple Crawler (SC), Evolutionary crawler (EC) and combined crawler (SC+EC) in AuthZee

5.2.2 Simple crawler vs Evolutionary crawler

Since all out experiments are only run for 24 hours, evolutionary crawler's non-deterministic behavior can affect its final results. Figure 5.1 demonstrates this behavior. However, AuthZee is designed to be non-terminating and in practice, it could be run as a background process indefinitely in which case the non determinism would not have much impact. Therefore, we pick the best results of evolutionary crawling from the nine experiments previously run during parameter tuning, to compare it's true potential with the simple crawler.

It is clear that evolutionary crawling outperforms simple crawling for all target apps except for Gitlab. This is because Gitlab contains many static help web pages which are quickly discovered by simple crawler, hence greater results. We note that simple crawler is strong in exploration without exploitation ability. And evolutionary crawler can exploit webpages to produce app state changing non-GET requests however due to it's slower speed, it is weaker in exploration. Thus, we run them together using Apache Kafka, which allows sharing of new pages discovered between the two kinds of crawlers.

Combining the simple crawler and evolutionary crawler brings both the advantages of the two crawlers, namely - exploration and exploitation. Simple crawler's low compute overhead allows fast exploration of static hyperlinks across webpages and evolutionary crawler's ability to perform automated browser interactions on a webpage UI enables exploiting a webpage to generate new app objects or discover resources hidden behind multi-step interactions. This is achieved by sharing new discovered URLs between the two crawlers. For instance, when evolutionary crawler generates a new app object, it's page URL is sent to simple crawler to quickly crawl for hyperlinks and any new URLs resulting from that simple crawl are sent to evolutionary crawler's priority queue to be crawled later. This teamwork yields higher results as evident with the hotcrp app where running evolutionary crawler individually yielded 2 not-GET requests while running combined crawler yielded 19 non-GET requests. Note that both experiments are run with same amount of compute resources like cpu, memory..etc.

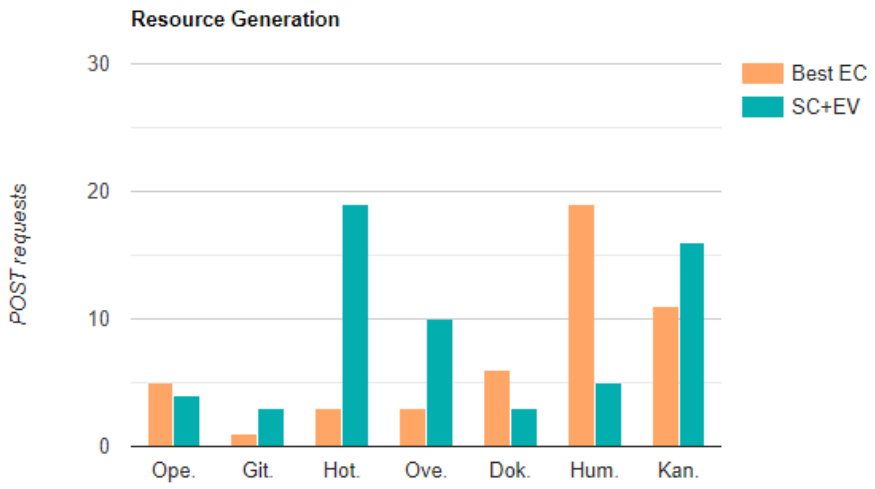

Figure 5.3: Illustrating object generation ability. Highest number of POST requests generated by Evolutionary crawler (EC) and combined crawler (SC+EC) for parameters s=5, p=10, i=25 AuthZee

Figure 5.3 compares the number of POST requests generated during best results of evolutionary crawling with the result of combined crawling (SC+EV). The result shows that combining the two crawlers outperforms evolutionary crawling except for Humhub and Dokuwiki. On investigating further, we see that the combined crawler produces lower POST requests for Humhub because it encounters many duplicate page requests. They don't appear as duplicates because they contain different input parameter and values, however they have the same pathname. For instance,

`http://hostname/directory/members?keyword=+HumHub`

`http://hostname/directory/members?keyword=+Digital+Native&page=1`

`http://hostname/directory/members?keyword=+French`

`http://hostname/directory/members?keyword=+Marketing`

These URLs are produced when user selects to view different categories of directory members as per the keywords, however they render the same page. This makes the evolutionary crawler exploit the same page multiple times and not have time to exploit other POST request producing pages. We can avoid this by only considering the pathname - `/directory/members` to determine if the request was seen before. However, this may exclude objects/pages that are actually referenced by different URL input paramters with different access restrictions. The combined crawler for Dokuwiki on the other hand did not have this problem. The lower result can be attributed to the non deterministic behavior of its evolutionary crawler.

5.2.3 Blind evolutionary crawling

Next, we perform blind evolutionary experiments in an HTML element tag agnostic way. In the evolutionary experiments thus far, the crawler knew all the interactable elements on the webpage before interacting with them, since we explicitly specified which element tags to look for when on a webpage, namely - <input>, <button>, <textarea>, <a>, <select>. So, we could hardcode the interaction as per the element, for instance clicking for <button> and <a> (hyperlinks) or typing for <input> and <textarea> elements. And the crawler would search for the sequence of these intractable elements that

	ope.	git.	hot.	ove.	dok.	hum.	kan.
page urls	6	11	6	1	2	35	1
all urls	18	30	7	3	2	53	1
filtered urls	5	24	2	2	0	33	1
non-GET	1	0	1	0	1	7	0

Table 5.8: Blind Evolutionary Crawling Results

generated a non-GET request. However, on some complex website UIs like in Facebook and Google, the interactive elements are wrapped in generic container tags without disclosing their basic HTML tags in the page HTML. For instance Facebook's Like button elements are wrapped in generic (division) tags. To test our evolutionary crawler approach over these types of UI, we make the evolutionary crawler to only look for generic HTML elements like tags when interacting with pages. This adds additional complexity since the crawler now also needs to search for the appropriate interaction for an element. Another challenge is that the crawler no longer knows if the chosen element is even interactive. Thus, the search space greatly increases. Anticipating this, we increased exploitation by running these experiments with a higher i=50 (number of evolutionary iterations per page). We set s=5, p=10 the same as before.

After running this crawler on all target apps for 24hrs, the results are shown in Table 5.8. Although lower than previous non-agnostic evolutionary experiments, we demonstrate that this evolutionary crawling approach is able to generate non-GET requests in a HTML tag agnostic way. Gitab did not produce any non-GET requests because it is relatively bigger than other apps with many interactive elements. It did however produce 24 filtered urls. Futhermore, Blind crawler could not generate non-GET request in Kanboard and overleaf. On inspecting their UI we learn that they sparsely use tags to enclose their elements. Furthermore, they use the elements as broad containers that contain of multiple interactive elements. By default the crawler performs the interaction (ex. click) on the center of the element, in this case the container element. If there is no interactive element in that part of container then the crawler mistakes it as a non-interactive element. However, for webpages with many interactive elements like Humhub social network app, we still are able generate and discover many requests.

5.3 Detection by Triad Tests

In this section, we evaluate our triad detection method on our target apps. The requests discovered during the crawling experiments are sent to triad detection.

5.3.1 Triad : Proof of Concept

Before running any detection experiments, we need to ensure that the triad test successfully differentiates between a response to authorized request and a response to an unauthorized request. Following is the proof of concept for triad testing.

The assumption of triad test is that userA is authorized and userB and userC are unauthorized. We will hold this assumption as our reference. Next, using the admin account of our target apps, we

Case #	Accessed Resource (R) ?			Expected Triad Test Result
	UserA	UserB	UserC	
1	Y	N	N	Pass
2	Y	Y	N	Fail
3	Y	N	Y	Fail
4	Y	Y	Y	Fail

Table 5.9: Triad Test Scenarios

App	URL pathname
Gitlab	/userA/testproject
Hotcrp	/2020f/settings/reviews
Dokuwiki	/requirements
Humhub	/s/private/
kanboard	/user/2/edit

Table 5.10: URLs of Triad Tests

configure the access controls for a resource R to create 4 test cases described in table 5.9. Case 1 is the proper configuration of access control where both userB and userC could not access R and thus must pass the triad test. While the rest of the cases must fail the triad test because in these cases, users B, C are able to access Resource R which contradicts our assumption that userB and userC aren't authorized to access R, thus implying that R is vulnerable to unauthorized access.

Table 5.10 lists the resources that were used for the triad tests. For gitlab, we created a new private project called 'testproject' and controlled its access from userB and userC through admin account. For Hotcrp, we used the admin page referenced by path - '/2020f/settings/review', where we set users as PC chair or regular members to control access to the admin resource. Similarly, we used the requirements page in Dokuwiki to test triad detection. For Humhub, from admin account, we created a private space called 'private' and added/removed users in that space to allow or restrict its access. Finally, in Kanboard we used userA's edit profile page and assigned users B,C administrator roles to allow access.

We perform these tests on our target applications and the results are shown in Table 5.11. Note that Openstack and Overleaf apps did not allow tweaking access controls and therefore we could not include a triad test result for them. However, similar behavior is expected as triad tests are not app logic dependant.

We observe that the results for case 3 and 4 in the apps - Dokuwiki, Humhub and Kanboard do not match with the expected triad test results. Although their similarity ratio difference (simBC - simAB) is small, it is greater than our chosen MF (0.001). This leads to such cases not being detected or false negatives. We plot these results in Figure 5.4 to better visualize this case. We must increase the *MF* to include these cases.

We infer that case 3 and 4 are brittle because userC no longer receives an unauthorized response. The triad tests don't need MF tuning when we have a ground truth (userA's good response) and a ground falsity (userC's 'not authorized' response). However, when userC also recieves a good response, it becomes difficult to accurately differentiate between the ground truth and falsity, or to determine

App	case#	sim AB	sim BC	(simBC-simAB)	Triad Test Result
Gitlab	1	0.003479366354	1.0	0.9965206336	Pass
	2	0.6120573718	0.004917367353	-0.6071400045	Fail
	3	0.003328761166	0.002403023805	-0.0009257373606	Fail
	4	0.960496144	0.9556748007	-0.004821343288	Fail
hotcrp	1	0.1047441065	0.477678804	0.3729346975	Pass
	2	0.998651416	0.1079576434	-0.8906937726	Fail
	3	0.1028527683	0.09988066557	-0.002972102763	Fail
	4	0.9985515209	0.9984516258	-0.00009989511013	Fail
Dokuwiki	1	0.6947880473	0.992728303	0.2979402557	Pass
	2	0.9969765269	0.7016678249	-0.295308702	Fail
	3	0.7023627519	0.7068797776	0.004517025712	**Pass**
	4	0.9945249326	0.9919001012	-0.002624831366	Fail
humhub	1	0.4965595131	0.9819308393	0.4853713262	Pass
	2	0.9937909119	0.4917125354	-0.5020783765	Fail
	3	0.471733793	0.4950326308	0.0232988378	**Pass**
	4	0.9729133651	0.9707634671	-0.002149898061	Fail
kanboard	1	0.389886709	0.9990551494	0.6091684404	Pass
	2	0.9606900144	0.3657485767	-0.5949414377	Fail
	3	0.389886709	0.3811355593	-0.008751149719	Fail
	4	0.9607626985	0.9986527571	0.03789005856	**Pass**

Table 5.11: Triad Test Results

if userB's response was more similar to userA's or userC's, thus requiring *MF* tuning. By tuning the *MF*, essentially we are performing linear classification in it's simplest form, between vulnerable and non vulnerable classes of URLs and the *MF* acts like a decision boundary as shown in Figure 5.4.

Triad tests infer whether the unauthorized request was successful, by comparing the web page responses. And majority of content in the web pages consists for the html structure. Therefore, this works for web apps that have similar response page structure (HTML/JSON) for access requests by users with same privilege level. However, for web pages that have same page structure for access requests by different users would result in false positives. For example, a home or account dashboard page URL accessed by different users will have the same page structure with different user data. Such web pages are generally the static pages accessible by all users and we mitigate these false positives by filtering them in Public Filter module.

5.3.2 Minimizing False Positives

We observe the results in table 5.12 and 5.13 to evaluate the impact of the Public Filter module in mitigating false positives. These tables show the false positives encountered when crawling for horizontal and vertical privilege escalation or when userA was a regular user and admin user respectively. The results not only have low false positives but also show the amount of false positives prevented by using the Public filter. For instance, in table 5.12, gitlab produces 1,510 request URLs of which only 93 are user specific. Thus, it prevents 1417 potential false positives.

We initially identified two main reasons for false positives. First, the public filter module missed filtering out some public URLs. This was because in the public filter module, we only used the simple crawler approach to discover URLs with userB's session. This allowed some public resources hidden

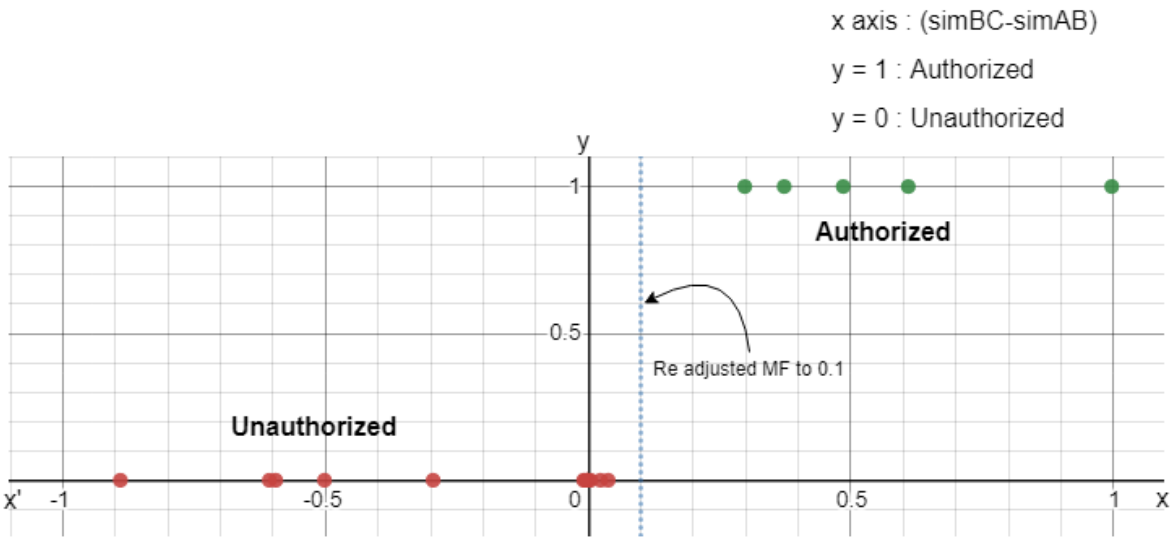

Figure 5.4: Triad Results Plot

behind multi-step user interactions to be missed and therefore not be filtered out in the public filter module. We mitigated this by using the combined crawler in the public filter module. Since evolutionary crawler is non-deterministic, there may still be a public URL that slips through the filter causing a false positive.

Second, the presence of anti-CSRF tokens in URL's input parameters prevents them from being replayed successfully during triad detection. This results in all three users receiving identical error responses, causing the triad detector to falsely report a vulnerability. We solved this issue by storing the userA's legitimate response during the crawling. And, in the detector module, a URL is only replayed for userB, userC and compared to userA's response received during crawling. This works when the anticsrf token is sent through the cookie, since we swap cookies in the request header with the cookies received after a fresh login of userB, userC. But, if the anticsrf token is sent in some other part of request, for example - the request body, it will not allow us to successfully test access control as both userB, userC will always receive failure response since we do not swap the tokens in the request body with their valid anti-csrf tokens. However, it will not generate a false positive.

Note that if the default access control configuration of the app allows userB and userC to access userA's resources, then again we have false positives. For instance, dokuwiki's default access control lists allows all users to access all pages [1], even if the users belong to different groups. Therefore, we see all filtered urls reported as vulnerable. Setting more restrictive access controls avoids this problem. Thus, selecting right triad of users is important. This does require some human intuition based on slight understanding of the app logic and so randomly picking three users may not be a good idea.

Another case of false positive is encountered when the HTML content is too large and the user specific data is sparse or the error message for the unauthorized request in the HTML response is very small. This can cause the (sim_ratioBC - sim_ratioAB) difference to be lesser than MF thus reporting as vulnerability. Theoritically, this shouldn't be a problem if we have $MF = 0$ but in practice, we need to set it to some small positive value to allow for compute errors. The difflib package used to calculate

	ope.	git.	hot.	ove.	dok.	hum.	kan.
page urls	7	1444	86	3	89	51	6
all urls	46	1510	109	5	117	67	6
filtered urls	0	93	23	5	11	9	4
false +ves	0	6	0	0	11	2	0

Table 5.12: False positives during simple crawling for horizontal privilege escalation

	ope.	git.	hot.	ove.	dok.	hum.	kan.
page urls	7	675	106	3	107	50	11
all urls	47	761	113	5	113	62	11
filtered urls	3	207	61	5	5	14	9
false +ves	0	4	0	0	0	0	0

Table 5.13: False positives during simple crawling for vertical privilege escalation

similarity ratios, supports 16 places of decimal when calculating similarity ratios. So *MF* in practice cannot be less than 10^{-16}. Therefore adjusting the *MF* mitigates these kinds of false positives.

5.3.3 Testing different levels of Authorizations

Running the triad detector with different triads of users can test for different privilege levels of IDOR vulnerabilities. For instance, to test for horizontal privilege escalation, we pick a triad of users with the same privilege level - userA, userB, userC = all regular users and to test for vertical privilege escalation vulnerability, we pick a triad of users with uneven privilege level - userA = Admin and userB,userC = regular users. This is shown in table 5.12 and 5.13.

When selecting users for triad testing, userA must have the a privilege level higher or equal to the other two users. The simplecrawler and evolutionary crawler must be run with the first user's credentials (userA/Admin) and the other two users (userB,userC) must have the same privilege level and less than or equal to the first user's privilege level.

Moreover, this idea can be extended to test for authentication for a resource. If userB and userC cookies are replaced by empty strings, the triad test will now compare responses received to the authenticated userA with a non authenticated response. We do not explore these since our focus is on authorization.

5.4 Vulnerability Discovered

A vulnerability was detected in a popular web application. Since, a patch has not been released yet to fix it, we will not disclose the app name. The vulnerability was found in a AJAX call intended to be made by an admin user however, it was not access controlled, thereby allowing any registered user to access data like internal ID, name, username, account state, profile picture of all user accounts in the app. The vulnerability was reported and confirmed by the developers.

Chapter 6

Limitations

6.1 False Positives

Although we have the filter module to mitigate the false positives problem, it is not foolproof. We depend on the BFS crawling heuristic to find public or shared URLs. However, as we showed the BFS heuristic isn't very good at finding all resources and therefore it is possible to miss some public URLs hidden behind multi step user interactions. Hence, the missed public URLs are sent to the triad detector and contribute towards false positives. To improve the filtering we could use the evolutionary heuristics for the twin crawler however, this still will not guarantee complete elimination of false positives. Following this thought, there could also be false negatives or missed vulnerabilities if there is a logic error in the application where a link to a private object was mistakenly made available in a public page or other users' account page. In this case the twin crawler would access that mistakenly exposed private object and it will be considered as a shared resource, thus incorrectly filtering it out from the triad detection.

Our design assumes that the user who created the object is the owner of the object, meaning that they have the authorised access to view, modify or delete that object. However in some apps, the creator may not have permission to modify or delete the object after creation. This is not a problem unless the website user interface (UI) renders the elements to modify/delete that object and upon interacting (ex. clicking) on it, a generic unauthorized message/page is displayed. This kind of user experience (UX) design will confuse the triad detector and may generate false positives as the ground truth itself is unauthorized. We perceive this as an improper UX design of the app. Our system tries to infer developer intention from the web app's UI, like a first time human user would. So, if an element to modify or delete a object is displayed on the screen, then the user would assume to have access to that. However, for a proper UX or frontend design, the elements in a page that try to perform unauthorized requests must be disabled or hidden, which does prevent such false positives.

6.2 Input Validations

For some inputs, there exists client side regex validations like the input value must be a number, email, or some other regex. This will prevent the evolutionary crawler from generating POST requests or creating objects on that page even if it found the right sequence of element-interactions. Moreover, other validations like captchas, no CAPTCHA reCAPTCHA(I'm not a robot check mark), image classification tasks...etc are also not supported by our design. However, these validations usually exist outside the user account space when the user is not logged in or when registering a user account. Although one approach to handle this could be to try out a set of pre-created input values during the evolutionary crawler. For instance when typing text into some element, we randomly select one of the values in our input set - sample email - 'AuthZee@test.com', numbers - '0', '-1', '100', date - '2021/01/01', random string that satisfies most password regex validations - 'aA1!aaaaaa'. Furthermore, there exists research on automatically registering to create user accounts like the cookie hunter [32] and it would be interesting to see the results after combining these two approaches.

Chapter 7

Related Work

Authorization bugs have been prevalent for a long time and a number of papers in the past have proposed techniques to detect these vulnerabilities. Since, in this paper we leverage concepts such as automated browser crawling and evolutionary algorithms with novelty search, further paragraphs will also review past papers in the aforementioned areas. We will discuss previous blackbox approaches where source code is not known, greybox approaches where source code is not known but some other system related information is leveraged and whitebox (static analysis) approaches where the source code is known.

7.1 Existing black box web vulnerability scanners

We explore different blackbox security scanners as these tools are commonly used by security researchers to scan web application for security vulnerabilities.

Demesa [29] evaluated three open source blackbox security scanners - w3af, OWASP ZAP and NIKTO for finding IDORs. w3af is a penetration testing tool for attacking and auditing web applications. Similarly, NIKTO is a command line vulnerability scanner used to scan webservers. OWASP ZAP short for Open Web Application Security Project Zed Attack Proxy, is another penetration testing tool that automatically scans the whole application and shows vulnerability results in an intuitive Graphical User Interface (GUI). Although these tools are highly efficient in scanning, Demesa shows that they are not able to find IDOR or any broken authorization related vulnerabilities.

Porat et al. [49] propose an automated authorization enforcement detection (AED) tool that enables system administrators to detect authorization vulnerabilities on their websites. Their solution is similar to semi-automated techniques like Burp with Autorize extension, however they also support CSRF-token enabled websites and preform a deeper analysis than solely comparing HTTP response codes and lengths. Their approach requires an admin to manually surf through the website and AED intercepts the requests made as a result of the web surfing. The responses containing cookies is stored for further analysis. Then the recorded requests are resent with different cookies and the response string is matched with original response string. An breach is reported if the similarity score is a greater than 95%. However, it cannot infer if a request was intended to have shared access and therefore will encounter many false positives.

7.2 Other black box techniques

The cookie hunter paper by Drakonakis et al. [32] describes an automated black-box approach for detecting authentication and authorization flaws by analysing the session cookies after user login. They focus on cookie hijacking attacks where cookies not protected with the httpOnly flag can be stolen using Javascript code through a cross-site scripting (XSS) attack. The paper showcases automatic signup and login to webapps and then inspect the cookie returned after login to determine if it is flawed or hijackable. This gives insight on cookie related weaknesses such as if unprotected cookie exposure or JS cookie stealing is possible. Their custom browser automation tool - XDriver audits web apps without needing any app specific knowledge. They start by BFS crawling target app link to collect pages with forms that contain account related keywords like login, signin. Extracting relevant elements from these pages like forms, they automate account creation process by using a manually curated set of regular expressions and python Faker package to input information as per the input validation constraints of the form elements. They perform various checks to determine if the signup was successful. For instance, checking if the sign up form element is still displayed or if an email is received. Next, they visit the login link previously collected to auto login. If traditional account generation process fails, they try Single Sign On (SSO) to sign in using their Facebook and Google credentials. On successful login, the session cookies received from server are analyzed by observing attributes like secure and httpOnly. For instance, if none of the cookies have these attributes, they can be stolen leading to session hijacking. Further cookie analysis is performed to check if excluding those cookie sets from browser's cookie jar allows user to stay logged in. With their automated approach, they were able to audit 25K domains. They discovered that over 10K domains exposed authentication cookies over unencrypted connections and over 5K domains did not protect authentication cookies. Their study demonstrates that basic cookie protections are absent or incomplete in many applications

AuthScope by Zuo et al. [56] automatically executes a mobile app to detect vulnerable authorizations. They perform differential traffic analysis to recognize the protocol fields in the request structure which are then automatically substituted and checked for correct authorizations in server response. They develop a targeted dynamic activity explorer to automatically log in the app and explore the app activities in a prioritized depth first search approach to get the post-authentication messages. This works well for mobile apps due to the layered structure of the in app activities. They also perform app login with Single Sign On (SSO) to auto login and explore post-authentication messages. They take two legitimate users, compare their responses received for the same request and identify the differences as protocol fields that take user specific inputs. Then these fields are substituted such that the modified value has a small euclidean distance. After sending the request with substituted protocol fields, if the server responds with private messages of a different user, the request is labelled as vulnerable. Leveraging SSO, they are able to test 4,838 mobile apps from Google Play and detected 597 0-day authorization vulnerabilities in 306 apps.

AuthScope performs static view exploration to discover requests and does not handle dynamic object generation. Therefore, AuthScope only checks unauthorized reads and does not support unauthorized write attacks. They prune public activities/interfaces accessible prior to login. But they assume that all post login resources are private and therefore encounter false positives for resources that are intended to be shared or public.

Doupe et al. [31] presented a way to infer the internal state machine of an app by crawling and observing if a response to a previously made request changed. This state awareness is used to guide a blackbox web application vulnerability scanner, where the user-input vectors obtained from crawling are fuzzed for security flaws. However, they do not support AJAX requests, multi-step javascript interactions to create app objects, and crawling hidden web content. Moreover, maintaining internal state is not scalable for larger complex applications.

7.3 Grey box

In gray box techniques, some information (other than source code) regarding the application is required. We review past work that requires app details like OpenAPI specification and Network Traffic Logs.

7.3.1 OpenAPI specifications

Yelp's Fuzz-lightyear is a framework designed to automate IDOR discovery through stateful fuzzing [42]. It leverages the Swagger or OpenAPI specifications of a web application, first proposed in the RESTler paper by Atlidakis et al. [24]. Swagger specs are machine readable code that list API endpoints and their required parameters for successfully making a request to that endpoint. Originally designed for the purposes of auto generation of documentation of the APIs, Atlidakis et al. show the use of these specifications to automatically determine producer-consumer relationships between the API endpoints. They then generate chains of such requests or sequences of requests that allows exploration of a much deeper state of code, due to its statefulness. The idea is to call these sequences of requests multiple times while modifying the input parameters, hoping to encounter a server failure message. This process of running a program multiple times with different inputs is called fuzzing and the key contribution of RESTler was stateful fuzzing of REST APIs which by themselves are stateless. RESTler was designed to be a generic bug detecting tool. Therefore, it can only detect the bugs that cause the app server to respond with a HTTP 500 (Internal Server Error) code and cannot detect if a purposely formed malicious request succeeded.

However, the fuzz-lightyear project leveraged RESTler's technique to focus on authorization based bugs. The fuzz-lightyear calls these sequences with two different user sessions. And attempts to call the final request of the sequence with another user's session. The intention is to check if a resource of the state created by one user can be accessed by another user. To clarify here, the state is created by the subsequence of requests excluding the final request.

However, fuzz-lightyear does not fully automate the authorization bug finding process. For instance, a resource that is intended to be publicly accessible by everyone needs to be explicitly provided before running fuzz-lightyear and for a complex app, this can be a lot of work. Moreover, fuzz-lightyear also needs additional information regarding the generation of objects in case the server relies on some other microservice outside the scope of it's API specifications to create that object. Furthermore, these approaches require OpenAPI specifications of the app and not all app creators maintain or release such specifications. And manually creating such specifications for each app is time consuming.

7.3.2 Network traffic Logs

The approaches that infer access control policies from the network traffic or access Logs also have been proposed in the past. Karimi et al. [39] and Marinescu et al. [43] leverage unsupervised machine learning algorithms to learn the access rules from the server logs or other data manipulation patterns. After learning new rules, their models detect the activity outside the rules as unauthorized.

Although an automated technique that doesn't require human intervention after the initial setup, it needs a high amount of user traffic to be effective. Furthermore, most apps are constantly updating or adding features. Thus, in practice, after every update, it will produce false positives so there will be a window of time for which the model can not be applied as it will be in the process of learning the access rules.

Pellegrino and Balzarotti [48] propose another technique to find logic flaws in web applications. They also use network traces to model inter-dependencies between requests and generate logical patterns or sequences. These logical patterns also imply authorizations. Contradicting patterns are then tested to check for logic and authorization flaws. However, these logic patterns are specific to ecommerce web apps so their design is not fully application agnostic. Furthermore, a human presence is required to detect if a logic property was broken.

7.4 Whitebox Static Analysis

Tyagi and Kumar [53] evaluate static web application vulnerability analyses tools - OWASP WAP and RIPS on detecting intentional vulnerabilities in DVWA - damn vulnerable web app, bWAPP - buggy web app, which are deliberately insecure web application. Their results show that such tools perform quite well for automatically detecting vulnerabilities like Cross Site Scripting (XSS) and Remote File Inclusion (RFI) but cannot detect IDOR. Furthermore, these tools requires access to source code and can only be used for apps built in PHP language supported by the tools.

Zhu et al. [55] propose a hybrid interactive static analysis technique to mitigate access control vulnerabilities by requesting input from developers. Their solution is designed to be deployed during the development process as part of a code editor or Interactive Development Environment (IDE). Their approach inputs Abstract Syntax Trees of the web application and a set of security sensitive operations (SSO). When a SSO is identified, the developer is notified to highlight the code containing the intended access control logic. Then static analysis techniques are used to detect access control vulnerabilities based on the information provided by the developer. Since, this technique receives information directly from the developer, it is able to outperform commercial static analysis tools. However, this approach frequently involves extra human effort which can prevent it's adoption in the developer community. Furthermore, they focus on finding code patterns that may cause unauthorized accesses rather than unauthorized accesses themselves. Therefore, these static analysis tools generally suffer from high false positives.

Demesa [29] presents evaluation for two more web application static analysis tools - SonarQube and Checkmarkx. The results show that these tools cannot automatically find any IDOR vulnerabilities from source code of web apps.

7.5 Evolutionary Algorithms with Novelty Search

In the past, evolutionary or genetic algorithms have been used to solve Travelling Salesman Problems (TSP) and their many variations [36]. The general idea revolves around finding an optimal order or sequence of visits that the salesman must make such that the total distance travelled or time taken is minimized, given a list of cities to visit. We can draw parallels from this problem to the evolutionary crawling problem. Instead of sequences of city visits, we have sequences of page element interactions and instead of a single optimization objective like distance travelled, we have a bit more complicated reward system. This provided the initial intuition that genetic algorithms could be applied to the crawling problem.

Adding novelty search in our evolutionary crawler was inspired from work by Lehman and Stanley in their Abandoning Objectives paper [41]. They demonstrate the importance of searching for new behaviours by showing how novelty search significantly outperforms objective-based search for maze navigation problems. Although maze navigation isn't directly applicable to web application navigation, their evaluation on maze problems helps us understand the behavior of evolutionary algorithms and their tendency to get stuck in local optimum. For a non convex objective function, solely focusing on objectives can lead to dead ends that mask themselves as the next logical step with a greater objective score than previously seen but non of the subsequent steps from that step lead to better objective score, hence a dead end. Lehman and Stanley show than the search for novelty can circumvent this deception by evaluating the novelty search approach for a three-dimensional biped robot simulation. The results demonstrate that the novelty search evolved robot controller walks almost double the distance than pure objective fitness based controllers for the same search time. Thus, they conclude that search for novelty can outperform search for objectives in complex problems where formally defining an objective function is hard. Such problems must be guided by diverse range of information rather than single explicit objective.

This search for novel behavior has an application in software testing since we would like to test a system for all kinds of novel inputs as they are more likely to find software bugs in the system. Attwood et al. [25] throw light on the divergent nature of novelty search and how that can be useful for test data generation within web security.

They show that in the past EAs have been applied in areas like SQL injection, XML injection, Cross Site Scripting (XSS), Denial of Service (Dos), CAPTCHA generation and spam email. From their survey, they conclude that much of past applications of EAs in web security haven't explored the underlying EA and thus there exist an untapped potential. They mention the potential of EAs with novelty search, originally proposed by Lehman and Stanley [41], in Web Security as a way to generate test data within web security. This is possible because novelty search does not strictly adhere to the objectives, allowing the algorithm to explore new regions in the search space. This is very useful as it can be used to automatically generate a diverse test data set. AuthZee's evolutionary crawler applies this concept to generate sequences of UI interactions that result in frontend requests to the web server.

Chapter 8

Conclusion

We have presented a novel approach to automatically discover resources and detect authorization vulnerabilities in a web applications. Our tool, AuthZee leverages an evolutionary algorithm to generate new objects during crawling and leverages triad testing to detect authorization vulnerabilities. We showed that evolutionary crawling approach outperforms traditional BFS approach as it is able to perform multi-step interactions with the frontend, thus reaching more resources. Furthermore, combining the traditional simple crawling technique with evolutionary crawling technique resulted in even better results. We also successfully mitigate the high false positive problem that existing automated authorization vulnerability detection tools face.

AuthZee takes three user account credentials and instructions to auto login to the target web application. We show that by selecting user credentials with different privilege levels, we can search for different types of privilege escalating authorization vulnerabilities.

8.1 Future Work

In our design, the evolutionary crawler only performs user interactions like click, typetext, file upload, selecting option. This can be extended to all possible interactions that testcafe supports like mouse hover, pressing specific key or combination of keys, mouse right click, drag..etc however this will increase the evolution search space and likely take more number of iterations to create user objects. Thus, increasing total evolution time. Further, AuthZee's evolutionary crawler was implemented using basic genetic algorithm strategies. Different specialized strategies for selection and mutation can be tried to further improve discovery and generation power of the crawler.

Another area to explore is the scalability of AuthZee. Multiple evolutionary crawlers can be run concurrently to horizontally scale AuthZee. Kafka allows multiple consumers to read from one queue. Therefore, multiple crawler can simultaneously interact with multiple pages at any given time, thus increasing crawling throughput. This is possible due to the modular design.

Current design of AuthZee does not support IDOR attacks to bypass business logic of application as it requires a higher level of business logic knowledge to determine what values the input parameters must be changed to violate the logic. Future attempts can be made to customize AuthZee to specific

app logics to support detection of these kinds of vulnerabilities.

Furthermore, AuthZee's evolutionary crawler has additional applications other than finding IDORs. The ability to generate objects and record request-response logs can be used as a seed for website fuzzers and web vulnerability scanners. Futhermore, tools like Restler can use this seed to fuzz websites that do not have OpenAPI specifications.

Bibliography

[1] Access control lists (acl)s. URL: https://www.dokuwiki.org/acl.

[2] Ajax introduction. URL: https://www.w3schools.com/js/js_ajax_intro.asp.

[3] Apache kafka. URL: https://kafka.apache.org/.

[4] Authmatrix - portswigger. URL: https://portswigger.net/bappstore/30d8ee9f40c041b0bfec67441aad158e.

[5] bitnami/dokuwiki - docker image — docker hub. URL: https://hub.docker.com/r/bitnami/dokuwiki/.

[6] Burp suite - application security testing software - portswigger. URL: https://portswigger.net/burp.

[7] Cross-browser end-to-end testing framework — testcafe. URL: https://testcafe.io/.

[8] Devstack — devstack documentation. URL: https://docs.openstack.org/devstack/latest/.

[9] Github - kohler/hotcrp: Hotcrp conference review software. URL: https://github.com/kohler/hotcrp.

[10] Github - overleaf/overleaf: A web-based collaborative latex editor. URL: https://github.com/overleaf/overleaf.

[11] gitlab/gitlab-ce - docker image — docker hub. URL: https://hub.docker.com/r/gitlab/gitlab-ce.

[12] kanboard/kanboard - docker image — docker hub. URL: https://hub.docker.com/r/kanboard/kanboard.

[13] mriedmann/humhub - docker image — docker hub. URL: https://hub.docker.com/r/mriedmann/humhub.

[14] Owasp zap. URL: https://www.zaproxy.org/docs/desktop/start/features/ascan/.

[15] Screenshots and videos — advanced guides — guides — docs. URL: https://testcafe.io/documentation/402840/guides/advanced-guides/screenshots-and-videos#record-videos.

[16] M6: Insecure authorization, 2016. URL: https://owasp.org/www-project-mobile-top-10/2016-risks/m6-insecure-authorization.

[17] Owasp top ten, 2017. URL: https://owasp.org/www-project-top-ten/.

[18] Github - yelp/fuzz-lightyear, 2019. URL: https://github.com/Yelp/fuzz-lightyear.

[19] Owasp api security project, 2019. URL: https://owasp.org/www-project-api-security/.

[20] Autorize - portswigger, 2020. URL: https://portswigger.net/bappstore/f9bbac8c4acf4aefa4d7dc92a991af2f.

[21] Choose between traditional web apps and single page apps — microsoft docs, 2020. URL: https://docs.microsoft.com/en-us/dotnet/architecture/modern-web-apps-azure/choose-between-traditional-web-and-single-page-apps.

[22] Burp scanner - portswigger, 2021. URL: https://portswigger.net/burp/documentation/scanner.

[23] Web Security Academy. Access control security models. URL: https://portswigger.net/web-security/access-control/security-models.

[24] Vaggelis Atlidakis, Patrice Godefroid, and Marina Polishchuk. Restler: Stateful rest api fuzzing. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 748–758, 2019. doi:10.1109/ICSE.2019.00083.

[25] Sam Attwood, Wanpeng Li, and Rupak Kharel. Evolutionary algorithms in web security: Exploring untapped potential. In *2020 12th International Symposium on Communication Systems, Networks and Digital Signal Processing (CSNDSP)*, pages 1–6, 2020. doi:10.1109/CSNDSP49049.2020.9249521.

[26] bugra. Idor when editing users leads to account takeover without user interaction at crowdsignal, 2020. URL: https://hackerone.com/reports/915114.

[27] Raj Chandel. Beginner guide to insecure direct object references (idor), 2017. URL: https://www.hackingarticles.in/beginner-guide-insecure-direct-object-references/.

[28] Data and Vulnerability Management Analysis. The rise of idor — hackerone, 2021. URL: https://www.hackerone.com/data-and-analysis/rise-idor.

[29] Ephrem Getachew Demesa. Implementation of a hands-on attack and defense lab on insecure direct object references. 2018.

[30] Iryna Deremuk. Web application architecture: A guide through the intricate process of building an app — litslink blog, 2021. URL: https://litslink.com/blog/web-application-architecture.

[31] Adam Doupé, Ludovico Cavedon, Christopher Kruegel, and Giovanni Vigna. Enemy of the state: A state-aware black-box web vulnerability scanner. In *21st USENIX Security Symposium (USENIX Security 12)*, pages 523–538, Bellevue, WA, August 2012. USENIX Association. URL: https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/doupe.

[32] Kostas Drakonakis, Sotiris Ioannidis, and Jason Polakis. The cookie hunter: Automated black-box auditing for web authentication and authorization flaws. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, CCS '20, page 1953–1970, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3372297.3417869.

[33] CDW Experts. 9 flaws of automated web application vulnerability scanners, 2020. URL: https://expertswhogetit.ca/security/9-limitations-of-automated-web-application-vulnerability-scanners.

[34] Karan Gandhi. Authentication & authorization in web apps, 2020. URL: https://blog.jscrambler.com/authentication-authorization-in-web-apps/.

[35] hunt4p1zza. [critical] insufficient access control on registration page of webapps website allows privilege escalation to administrator, 2019. URL: https://hackerone.com/reports/796379.

[36] K. Ilavarasi and K. Suresh Joseph. Variants of travelling salesman problem: A survey. In *International Conference on Information Communication and Embedded Systems (ICICES2014)*, pages 1–7, 2014. doi:10.1109/ICICES.2014.7033850.

[37] inhibitor181. Shopify disclosed on hackerone: Idor [partners.shopify.com] - user..., 2017. URL: https://hackerone.com/reports/243943.

[38] jon_bottarini. Idor via internal_api "users" endpoint, 2017. URL: https://hackerone.com/reports/349291.

[39] Leila Karimi, Maryam Aldairi, James Joshi, and Mai Abdelhakim. An automatic attribute based access control policy extraction from access logs. *CoRR*, abs/2003.07270, 2020. URL: https://arxiv.org/abs/2003.07270, arXiv:2003.07270.

[40] Issie Lapowsky. The facebook hack exposes an internet-wide failure, 2018. URL: https://www.wired.com/story/facebook-hack-single-sign-on-data-exposed/.

[41] Joel Lehman and Kenneth O. Stanley. Abandoning Objectives: Evolution Through the Search for Novelty Alone. *Evolutionary Computation*, 19(2):189–223, 06 2011. doi:10.1162/EVCO_a_00025.

[42] A. Loo. Automated idor discovery through stateful swagger fuzzing, 2020. URL: https://engineeringblog.yelp.com/2020/01/automated-idor-discovery-through-stateful-swagger-fuzzing.html.

[43] Paul Marinescu, Chad Parry, Marjori Pomarole, Yuan Tian, Patrick Tague, and Ioannis Papagiannis. Ivd: Automatic learning and enforcement of authorization rules in online social networks. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 1094–1109, 2017. doi:10.1109/SP.2017.33.

[44] mirhat. Ability to add arbitrary images/descriptions/titles to ohter people's issues via idor on getrevue.co, 2021. URL: https://hackerone.com/reports/1096560.

[45] Seyedali Mirjalili. Genetic algorithm. In *Evolutionary algorithms and neural networks*, pages 43–55. Springer, 2019.

[46] Lily Hay Newman. Google+ exposed data of 52.5 million users and will shut down in april, 2010. URL: https://www.wired.com/story/google-plus-bug-52-million-users-data-exposed/.

[47] OPTASY. 5 automation testing tools for web applications for 2020 — medium, 2020. URL: https://medium.com/@OPTASY.com/the-5-best-automation-testing-tools-for-web-applications-that-you-could-use-in-2020-powerful-and-23135826a569.

[48] Giancarlo Pellegrino and Davide Balzarotti. Toward black-box detection of logic flaws in web applications. In ISOC, editor, *NDSS 2014, Network and Distributed System Security Symposium, 23-26 February 2014, San Diego, USA*, San Diego, 2014. © ISOC. Personal use of this material is permitted. The definitive version of this paper was published in NDSS 2014, Network and Distributed System Security Symposium, 23-26 February 2014, San Diego, USA and is available at :.

[49] Ehood Porat, Shmuel Tikochinski, and Ariel Stulman. Authorization enforcement detection. In *Proceedings of the 22nd ACM on Symposium on Access Control Models and Technologies*, SACMAT '17 Abstracts, page 179–182, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3078861.3084172.

[50] John W. Ratclif and David E. Metzener. Pattern matching: the gestalt approach, 1988. URL: https://www.drdobbs.com/database/pattern-matching-the-gestalt-approach/184407970?pgno=5.

[51] Andrew N. Sloss and Steven Gustafson. 2019 evolutionary algorithms review. *CoRR*, abs/1906.08870, 2019. URL: http://arxiv.org/abs/1906.08870, arXiv:1906.08870.

[52] test. Access control vulnerabilities and privilege escalation — web security academy. URL: https://portswigger.net/web-security/access-control.

[53] Shobha Tyagi and Krishan Kumar. Evaluation of static web vulnerability analysis tools. In *2018 Fifth International Conference on Parallel, Distributed and Grid Computing (PDGC)*, pages 1–6, 2018. doi:10.1109/PDGC.2018.8745996.

[54] Yuchen Zhou and David Evans. Why aren't http-only cookies more widely deployed. *Proceedings of 4th Web*, 2, 2010.

[55] Jun Zhu, Bill Chu, Heather Lipford, and Tyler Thomas. Mitigating access control vulnerabilities through interactive static analysis. In *Proceedings of the 20th ACM Symposium on Access Control Models and Technologies*, SACMAT '15, page 199–209, New York, NY, USA, 2015. Association for Computing Machinery. doi:10.1145/2752952.2752976.

[56] Chaoshun Zuo, Qingchuan Zhao, and Zhiqiang Lin. Authscope: Towards automatic discovery of vulnerable authorizations in online services. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 799–813, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3133956.3134089.