403 Forbidden?
Ethically Evaluating Broken Access Control in the Wild

Saiid El Hajj Chehade*, Florian Hantke†, Ben Stock†
*EPFL, Switzerland, †CISPA Helmholtz Center for Information Security, Germany
*saiid.elhajjchehade@epfl.ch, †{florian.hantke, stock}@cispa.de

***Abstract*—In the context of web applications, the most prevalent vulnerability, according to the OWASP Top Ten, is broken access control. As access control (*AC*) is implemented on the server side, not having access to the code in live systems limits the ability of researchers to study improper *AC* issues in the wild. While several works have identified vulnerabilities in open-source applications deployed in researcher-controlled environments, the problem has not been studied in the wild because of ethical and legal considerations to not leak unknowing users' data. We address this gap in research and present the Variable Swapping Framework (VSF), the *first* ethically sound and scalable black-box framework to test for improper *AC* patterns in the wild. VSF's design is the result of our *in-depth ethical stakeholder analysis* and risk minimization while maximizing benefits in vulnerability detection. At its core, it relies on two accounts per site and swaps identifiers between them to access one account's resources with the other. On the 100 web apps we successfully tested, we find a total of 584 potential *AC*-sensitive HTTP endpoints, out of which 19 (across 7 sites) are exploitable flaws, which we disclosed responsibly.**

1. Introduction

Web applications are a cornerstone of modern society. From sharing pictures with friends, and buying products online, to filing tax returns, web applications handle a plethora of – often personal and sensitive – functionality. In any web service where users must have authenticated accounts, it is paramount that a user can only access (or manipulate) data that is explicitly meant for them. Naturally, while developing ever-growing web applications, developers might introduce numerous mistakes that allow attackers to circumvent access control (*AC*) protections. The most recent OWASP Top Ten of application vulnerabilities lists broken access control as the most widespread flaw [1]. While such attacks manifest widely in the non-academic community from actual leaks [2] to bug bounty programs [3], the academic community has not extensively researched such flaws *in the wild*, to get a wider picture.

The majority of the limited research into the space has been conducted on open-source applications [4, 5, 6, 7, 8]. Here, the researchers deploy the application within a controlled environment and add their own users to test out different roles and features using their proposed tools to find – mostly already reported – security issues. While this adds value to the security community by ensuring that these applications can fix newly reported issues, highly ranked websites rarely use such open-source software to run their applications. Instead, they rely on custom-developed applications, out of reach of any white- or grey-box approaches.

To test whether *AC* is improperly implemented, a researcher has to try to access data not owned by their user. In a recent example of the First American Financial Corp. [2], the attacker could simply change a sequential ID in a URL to access millions of title insurance records. However, accessing other users' data is both illegal and unethical, even if a "bona fide" researcher has the best intentions at heart. Hantke et al. [9] analyzed the ethical dilemma faced by researchers in search of such server-side vulnerabilities. They found that researchers shy away from doing such analyses on real websites to avoid legal and ethical problems. This chilling effect leads to the fact that – to our knowledge – virtually no research exists that studies broken *AC* in the wild.

Hantke et al. [9] also asked ethics experts about ways to conduct experiments to find broken *AC* in the wild in an ethically acceptable way. For such scans, the interviewed experts concluded that measurements could be conducted ethically if the study design ensured that no third-party data could be leaked or modified. Abiding by these guidelines allows researchers to close the research gap while remaining ethically sound in their measurements. In this work, we are the first to instantiate such an ethical framework for finding access control vulnerabilities in the wild.

We have designed a research pipeline that aims to measure the prevalence of *AC* vulnerabilities in the wild and on a large scale while balancing risk to users and website operators and benefits for society at every step of the design. First, we register two accounts with every website under test. We then implement a leader/follower approach to browsing the applications with the two accounts simultaneously, ensuring mirrored, meaningful, and consistent user interactions. Next, we filter network requests to identify HTTP endpoints with potential broken *AC*. For each such endpoint, we exchange resource identifiers between one user's request and the other. In case *AC* is not correctly implemented, this enables one user to access the sensitive data of the other. Importantly, given that the swapping of identifiers ensures access only to another account we control, we can detect vulnerabilities

without jeopardizing the privacy of real users of the given service.

In doing so, we find that out of the 100 sites we successfully tested, 15 suffer from 30 improper server responses to disallowed AC patterns, 19 of which lead to confirmed vulnerabilities in seven of the sites. We disclosed all vulnerabilities to the affected parties. Beyond the mere confirmation of the pervasiveness of this class of flaw in the wild, we more importantly also set guidelines for other Web researchers to follow in their own work. In particular, our research project aims to serve as an example of conducting real-world server-side vulnerability scans in the wild while adhering the ethical best practices.

To summarize, we make the following contributions:

- We conduct a thorough analysis and discussion of research ethics when scanning for server-side vulnerabilities, highlighting how to minimize risks and how to increase benefits to the entire ecosystem.
- We design and implement the Variable Swapping Framework for conducting ethically sound access control vulnerability detection in the wild. We make the code available to reviewers in https://anonymous.4open.science/r/improper-auth-0101.
- We report on the results of our experiments and highlight that out of the 100 sites we tested, seven are susceptible to at least one broken access control bypass, threatening users' privacy.

At the time of this writing, we have disclosed all vulnerabilities to the affected parties.

2. Background

In the following, we briefly outline the basic concepts of authentication in contrast to authorization.

Authentication & Credentials. Web authentication is the process by which a user/client proves to a server they are who they claim to be. Web authentication is a heavily studied subject and the security of various mechanisms like passwords, two-factor authentication, and more complex mechanisms are constantly re-evaluated [10, 11, 12, 13]. Once a user is logged in, their browser stores and transmits some information that allows the server to re-identify the user: through session cookies (initially delivered from the server to the client and transmitted with each request by the client), session identifiers in the URL, or through custom HTTP headers (e.g., in the form of JSON Web Tokens). The security of these sessions relies on the high entropy of the tokens and other cryptographic primitives that prevent brute-force attacks and guarantee that a valid session token is from the legitimate user [14]. From now on, we refer to any authentication token or parameter that proves user identity as *credentials*. Credentials are not restricted to session tokens but also include any information used to validate the **authenticity** of the request, i.e., the request comes from a legitimate user, even if websites do not require a login and explicit user identity. For example, ticket numbers used to access airline booking management, order codes for delivery services, etc. are *credentials* if they are the only identifier used to grant the client (browser) rights to access the private areas of web services. In our work, we assume the authentication is secure and consider the security issues of *credentials* and authentication out of our scope.

Authorization & Access Control. In simple terms, while **authentication** allows to answer "Who is the user?", **authorization** instead asks "Is the identified user allowed to do *X*?", with *X* being some website interaction. More formally, an authorization or access control (*AC*) policy is a set of rules that govern the *actions a* of *user/subject u* over *resource/object* `r`, i.e., given a tuple (u, r, a) the *AC* policy returns *allowed* or *disallowed*. For example, an *AC* rule on `amazon.com` could be "the *user* can only *cancel* their own *orders*" or expressed as `(user_id,order_id,cancel)` is *allowed* only if `creator(order_id) = user_id`. Defining an *AC* rule for every triplet is expensive, so system designers resort to more coarse-grained policies like *discretionary access control (DAC)* – where each user has privilege over resources they create to read, write, delete, and delegate – and the more popular *role-based access control RBAC* – where users get assigned global roles, and different roles have access to different permissions e.g., all users with role "teacher" can edit the grades page [15]. The HTTP specification features a corresponding status code to indicate the fact that access to a resource had been denied: `403 Forbidden`. However, not every server-side implementation adheres to this principle of communicating the error in the status code (see Section 7).

There are numerous reasons why *AC* checks might fail their purpose. The most common issue is a simple misconfiguration of an *AC* check, either by implementing a wrong check or forgetting the rule altogether. This is especially true for more complex systems, where generic systems (like RBAC in *Django*) are way too coarse to handle all rules. In *Amazon* for example, it is important that users only have access to their own orders, and such explicit checks should be implemented throughout all APIs handling resources. With the growing number of APIs, the multiplicity of resources, and the complexity of services offered, errors become more likely to occur in covering all *AC* edge cases. Another example of code logic flaws occurs when only the initial request in a business flow is checked for authorization while an attacker can exploit all the subsequent ones. Finally, services might base their *AC* mechanisms on long hard-to-guess IDs for private objects (e.g., order tracking IDs). If this identifier is leaked as discussed in Section 4, it immediately compromises requests relying on its' secrecy, as it would not be checked against the requesting user.

As outlined, broken *AC* issues are very common in the real world [1]. In addition to the earlier mentioned example of First American Financial Corp [2], a plethora of high-profile reports exist. For example, an *AC* flaw in Instagram allowed an attacker to access phone numbers and user details by abusing a contact import API endpoint [16]. In NordVPN, the change of a user ID could have leaked

Table 1. COMPARISON OF SERVER-SIDE SCANNING STUDIES ON BLACK-BOX ACCESS-CONTROL TESTING.

Study	# sites	In the Wild	Ethical to Run Live	In-page Interactions	Automations	New Vulns.
Ishida [5]	4	×	×	×	C,F,V	0
Chaleshtari [6]	2	×	×	✓	C,F,V	4
Rennhard [7]	7	×	×	×	I,C,F,V	0
Deepa [8]	5	×	×	✓	F,V	0
Our work	100	✓	✓	✓	A,F,V	19

*Pipeline Stages: **A**: account management, **I**: Defining interaction protocol, **C**: collecting requests, **F**: filtering requests and sending candidates, and **V**: flagging improper AC responses.*

the users' payment histories [17]. A similar issue had been detected for TikTok ads, which would have allowed deleting support tickets using an order id [18]. Lastly, an attack chain in Uber's backend portal would have leaked names, phone numbers, and other user-related information to anyone [19]. The impact of these vulnerabilities is significant, affecting both companies' finances and user privacy – especially given that these companies serve millions of users. These examples underscore the importance of research in this area.

3. Related Work

Server-side issues are challenging to study on a large scale. Except for open-source software, the backend code is typically not accessible to the researcher, limiting the visibility of the available server-side endpoints alongside implemented security checks, and hindering the use of static analysis tools. Instead, researchers must treat the server as a black box and learn about it by interacting with it dynamically, which naturally comes with its limitations. Without access to the backend code, researchers cannot guarantee coverage of the entire codebase and only guess code paths from available frontend code, leading to situations where they can only hypothesize the causes of certain behaviors [20]. If specific website behavior is restricted behind a login, experiments become even more complex and resource-intensive due to the need for authentication [21, 22, 23, 22]. In addition to the technical challenges, ethical considerations and legal risks to further complicate such studies [9, 24].

Specifically, testing for access control involves sending requests to access other users' pages and resources. Facing the daunting technical and ethical obstacles, the few prior studies touching on access-control vulnerabilities only evaluate their frameworks and techniques over a few open-source applications [5, 6, 7, 8]. On top, none of the prior work performs a proper ethical analysis of their frameworks. Deepa et al. [8] and Ishida et al. [5] implement frameworks where they collect requests from one user, automatically extract resource identifiers, modify the parameter values (at random or by appending a constant character, respectively), and send the request with the new parameter value. Using these tools on live systems carries the risk of potentially leaking an external user's data through the API endpoint with compromised *AC*. Both tools do not catch any new vulnerabilities. In contrast, Rennhard et al. [7] and Chaleshtari et al. [6] inadvertently reduce the framework's risk by including two users per visit and swapping parameters strictly between them. However, Chaleshtari et al. [6] requires a pre-written test case and manual authentication for the users, which is not scalable. Still, they managed to discover four new vulnerabilities in the two open-source apps they evaluated. Rennhard et al. [7] is the only work that strictly focuses on testing *AC* violations and chooses to fully automate the process from visiting the sites to flagging vulnerabilities. At each stage – visitation, filtering requests, generating candidates, and validating vulnerabilities – they implement a series of heuristics and pattern-matching techniques, similar to ours, to extract resource identifiers, and flag server responses to forged requests as vulnerabilities or not. However, they resort to only evaluating `GET` requests to prevent state-altering `POST` requests which would compromise their setup. Credentials and authenticated sessions are fed manually to their framework. Our work differs from prior work in two main aspects (Table 1): (1) we design all framework components with ethical considerations in mind, which allow us to (2) run directly on live sites and in the wild.

4. Threat Model

In this work, we are interested in threat models that exploit flawed server-side *AC* policies to gain unintended privilege over some or all user-accessible permissions or resources in a web app. We focus on attacks where the attacker has their own account and abuses *AC* policies to gain access to or manipulate a victim user's resources by altering request parameters [25]. The general pattern for such attacks means that an authenticated user gains access to a sensitive resource belonging to another user by exchanging the object identifier of their own resource with that of the victim. In essence, they swap out the genuine resource identifier with the targeted one. Such identifies may be changeable even by a layperson, e.g., by editing the URL bar or sending requests from the browser dev-tools [26, 27]. If such an authorization vulnerability exists, the attacker merely has to ascertain the object or resource identifier to launch the attack.

Within our work, we consider three *AC* attacker profiles of interest that vary from weak to strong:

Opportunistic Brute-force Attacker. This attacker aims to get access to any resource available through a web-app endpoint. They do not target a specific user. The attacker varies the resource identifier in the endpoint until the endpoint request returns a successful response, signaling a valid identifier, if the website endpoint has broken *AC* permission checks. The practicality of this attack depends on the complexity of the resource identifier. For example, if the resource here is a file and identified through its index in the database: `file1`, `file2`, etc.; then, the attacker only has to enumerate over possible indexes until they get a valid ID. In contrast, if files use high-entropy UUIDs, the attacker cannot find a pattern to guess valid file identifiers and resorts to an expensive brute-force attack. Facing an opportunist attacker, the security of the system depends both on *the entropy of the identifier* and the *validity of the AC policy*.

Opportunistic Reconnaissance Attacker. Similar to the brute-force attacker, this attacker also aims to compromise arbitrary users and their resources. Unlike the brute-force attacker, this attacker does not try random values of the object identifiers, but mines valid identifiers from public repositories. Examples of such repositories include web archives storing public pages containing various publicly visible user resources (e.g., user posts), discussion forums, GitHub issues where users could share logs and requests for debugging, public screenshots on social media, and leaked user URLs from browser extensions (e.g., URLscan.io [28]). Many methods already exist to automate the extraction of these identifiers [29]. In this setting, identifier entropy does not give any security guarantees. Instead, against this attacker, the security of the system depends both on the *public visibility of the identifier* and the *validity of the AC policy*.

Targeted Attacker. The strongest attacker, that we consider as baseline, aims to compromise a specific victim user. They can use any means possible to extract the relevant resource identifiers by using prior techniques or by interacting directly with the victim. This includes a large suite of established attacks to extract the vulnerable identifiers, such as social engineering (e.g., convincing the user to share certain URLs), over-the-shoulder attacks, etc. [30]. We distinguish this attacker from the account-takeover attacker, as the targeted *AC* attacker (1) requires less information to launch the attack (e.g., only resource identifiers) and (2) does not need to impersonate the victim user (by logging in with their credentials) to access/modify their resources – avoiding authentication security measures (e.g., e-mail notification on logins). In this setting, we cannot rely on the complexity or privacy of the identifiers to argue about the security of endpoints to *AC* violations; the only guard is implementing a correct *AC* policy.

5. Research Ethics

This project analyzes access control (*AC*) issues in live Web systems, i.e., investigating the behavior of server-side code. Server-side analyses are often considered methodologically challenging because these experiments leave the controlled laboratory environment and may impact systems involving various stakeholders. Disclosing vulnerabilities in live systems can also lead to challenging situations [31]. A recent paper [9] explored the feasibility of server-side experiments, including one related to *AC* experiments, in discussions with research ethics committee members. They concluded that if the experiment design is well considered, such experiments can be ethically acceptable.

In light of this, we conduct a stakeholder analysis as recommended by the Menlo Report [32] and Kohno et al. [31] to systematically evaluate the ethical implications of this project and come up with guidelines to adhere to throughout the project. Besides being a guide for responsible execution of our project, we also believe it can serve as an example framework for other projects, demonstrating how to incorporate ethical considerations in the experiment design.

5.1. Stakeholder Analysis

When investigating *AC* issues, three main stakeholders play a role: the end users of the website under analysis, the website owners or operators, and the research team itself.

End Users. End users are those who interact with the web service. Although they are only passively involved in the experiment, they still face potential risks. A poorly designed *AC* experiment could expose user data to the researchers or, modify them in the worst case. Activities that interact with the end user, such as unsolicited messages, can also negatively impact the end user. If the scans are too aggressive, they may also affect the performance and availability of the web service, leading to a negative end-user experience. We discuss measures to minimize and, ideally, eliminate these risks in the next section.

Despite these risks, we believe the benefit to end users is significant. Each vulnerability that is discovered by the research team and fixed by the website operators improves the web service's security and reduces the risks of end-user data being leaked. By minimizing the risks above, we believe the overall benefit to end users outweighs the potential downsides, as potential data leaks could easily affect all website users.

Website Operators. Unlike end users, website operators are actively involved in the experiment. For that reason, it must be discussed whether and how to seek their consent, and if not, to justify why [32]. From *AC* experiments, operators may be affected by the additional traffic, which can lead to higher costs. In the worst case, their website's stability suffers from the additional traffic. Creating fake accounts to test *AC* issues can add additional administrative overhead. Furthermore, exposed user data due to a poorly designed experiment may harm the operators in the form of stress and additional work they need to put into triaging the incident and informing the affected users. Public disclosure or naming the web service in publications could also pose a risk of reputational damage. The benefits for website operators are substantial. If a vulnerability is identified and disclosed responsibly, it helps operators understand and address *AC* weaknesses on their site. Ultimately, this helps to improve their service's security, reducing the risk of costly data breaches.

Researchers and Team Members. Finally, the research team conducting the experiment is also a stakeholder. While all team members have consented to conduct this research, it is still necessary to consider and minimize potential risks they may face. As with any web-based research, there is a possibility of encountering content that could be uncomfortable. Additionally, researchers may face legal threats from parties who do not approve the unsolicited web scans.

The benefit for the research team is eventually a contribution to the academic community through the publication of their findings, which will raise awareness and understanding of *AC* issues. Furthermore, like other end users, the

researchers benefit from any positive security improvements resulting from this work.

5.2. Ethical Guidelines

To address the list of potential risks identified in the previous analysis, we discuss each risk in the following and propose measures to incorporate into our framework for detecting server-side access control vulnerabilities.

Data Leakage and Manipulation. *End users* and *website operators* could both be negatively affected by leaked or manipulated data. In their study, Hantke et al. [9] interviewed (among others) research ethics committee (REC) members for major security conferences. As a result of their discussion with the research team, the REC members suggested that a good way to avoid exposing end-user data is to experiment only with accounts owned by the research team. This way, any potentially leaked or updated data would only affect the respective other researcher-owned accounts. The framework must thus ensure that it interacts solely with researchers-controlled accounts. While it is possible that scans may view publicly available user information, the framework ensures that no sensitive data is accidentally disclosed. In addition to the technical measures, all researchers who are involved in the vulnerability analysis agreed on a self-commitment declaration to not use the gained knowledge to act outside the experiment context.

Fake Accounts. As mentioned above, the framework ensures to only test with researcher-associated accounts. Due to the administrative overhead such fake accounts could impose on *website operators*, a balance must be maintained in the number of accounts created and required. We decided to create two accounts per site that have recognizable test account names.

Informed Consent. As mentioned earlier, it is essential to discuss whether and how to seek informed consent of the *website operators*. Ideally, researchers would request consent from every web operator whose website is tested. However, research has shown that even vulnerability notifications receive a very low response rate [33], and we expect even lower rates when asking for consent limiting the study to a very small and potentially biased scale. According to the Menlo Report [32], research that would otherwise be infeasible can be performed without an informed consent process but must be well justified. This is the case for our research, which aims to design a framework for a scalable study that is indeed infeasible if operator consent is required. Hence, we have chosen a study design in which we do not acquire operator consent. To ensure this design aligns with current research ethics, we consider ethics in every research stage to minimize risk for all involved parties and have obtained approval from our ethical review board (ERB).

To still give the operators the right to withdraw from our research [32], the framework should include information on how to withdraw (in our case, in an HTTP header) and use one fixed IP address that can be blocked.

Website Resources and Stability. An immense usage of resources or impairment of the website's stability due to web scans would have a negative impact on *end users* and *website operators*. To minimize this risk, the framework should only send a normal amount of valid Web requests, similar to those made by the typical user. Additionally, we decided that every request that is automatically sent with our framework needs to be successfully performed manually by one of our researchers in the form of visiting the site. For unexpected responses, e.g., a 500 Internal Server Error, during a later automated stage, there must be safeguards in place to react accordingly. In our framework, we see such responses in the database so that we can investigate the difference further and verify the page's availability.

To avoid additional costs or impacts on the website's stability with too many requests, the experiment framework should enforce request limits per site, ensuring that the generated traffic remains negligible compared to the already existing noise on the Web. We decided to limit the requests to 16 per HTTP endpoint.

Alarms. Same as with the resources, potentially triggered alarms could put stress on *web operators*. At the same time, any alarm that leads to the earlier detection of an existing vulnerability – even before we disclose it – is a good alarm, as the vulnerability should not exist in the first place. Such vulnerabilities, if exploited by adversaries, could lead to mandatory reporting under privacy regulations (e.g., GDPR), causing even more extra work or even financial penalties for the operators [9]. By ensuring that the scanning framework only uses researcher-controlled accounts and avoids leaking any sensitive data, the risk and mentioned additional work are minimized for the operators. Furthermore, a request limit implemented in the framework lowers the number of potential alarms.

Responsible Disclosure. The key benefit that *all stakeholders* get from this work is the identification and remediation of detected web security issues if disclosed properly. Therefore, a discloser strategy must be considered before the experiments begin and be balanced with the timeline of the experiment. Issues disclosed too early could influence measurement results (e.g., operators fix the same issue for multiple of their sites in our dataset), while issues disclosed later could raise the risk of exploitations by malicious actors. Therefore, we decided to prepare our disclosure communication during the execution of the experiments and send them for every verified vulnerability immediately after all scans are finished. The disclosure emails must contain enough resources and recommendations to minimize the additional work that web operators do.

Uncomfortable Content. To protect *research team members* from viewing content they feel uncomfortable with (respect for persons [32]), we ensure that team members have the option to skip a website at any time. In such a case, the lead researcher would review and evaluate the skipped site.

Legal Issues. Legal risks always exist for a *research team* conducting this type of research. We have carefully considered our experiment pipeline and are confident we do not violate any legal regulations, as our testing is only limited to two researcher-controlled accounts, thereby avoiding the classical brute forcing of IDs and parameters. One exception is creating test accounts, which may still violate some websites' terms of service. We accept this risk as appropriate, as we will not pursue further action if operators block or delete our accounts. To further minimize legal risk for individual researchers, we run all experiments from our institution's IP address rather than from private IPs.

Summary. Broken *AC* vulnerabilities can not only cause financial loss to the operators but also jeopardize the privacy of millions of users, as evidenced by the previously mentioned examples. As Hantke et al. [9] noted in their work, ethics committee members, as well as the Web operators, were generally confident about such kind of research when conducted responsibly. Additionally, we carefully studied and discussed the recommendations from relevant ethics research publications [32, 31], and requested feedback from our ethical review board (ERB). They concluded that "[t]here are no ethical concerns against the implementation of the proposal", however, suggested that the involved researchers sign a self-commitment declaration promising to not "use the gained knowledge to act outside the experiment context". Additionally, they mentioned that all involved researchers should be made aware about potential legal actions that vendors could take and the measures we have implemented to mitigate these to enable an informed decision about whether they wish to participate in the research. We have followed both these recommendations.

In summary, considering the significant risks associated with potential data leakages through improper access control, it is paramount to identify such flaws and inform the operators to have a positive outcome on the security of the entire ecosystem. Thus, we believe the potential risks to the stakeholders outlined within this section are significantly outweighed by the benefit stemming from finding such flaws; under the assumption that the research framework is designed in a way to minimize the risks.

6. The Variable Swapping Framework (VSF)

Without access to backend source code, evaluating whether web endpoints (GET requests to fetch pages and GET or POST API calls) implement sound *AC* in a black-box manner is not straightforward. First, studying *AC* issues is only relevant to websites with logged-in user accounts that can own private resources (e.g., profile pages, files, etc.). Second, challenging the server's *AC* policy requires attempting to request other users' resources within a user's session. Throughout this section, we refer to requests which which aim to find improper *AC* checks as *probing requests*. While successful requests indicate a faulty *AC* policy, they might cause an ethical violation by revealing other users' private data. As stated earlier, to avoid compromising external users in our study, we must only target users we control, meaning we need at least two users per website. As managing all the above by hand is not scalable, we design the VSF framework which automates most parts of the experiments, leaving only peripheral human involvement. To efficiently manage user accounts and instantiate authenticated website sessions – in a semi-automated fashion – we adapt the *Account Framework* by Rautenstrauch et al. [21] which distributes fresh logged-in sessions to our automated browser agents. To generate probing requests ethically, we collect pairs of requests from two fake users logged in to two browsers – a *leader* browser controlled by the researcher who performs usual user interactions and a *follower* browser that mirrors actions from the *leader*. We then introduce an automated pipeline to match request pairs and extract *AC* parameters (user credentials, resource identifiers, etc.). The VSF then constructs probing requests by only swapping the resource identifiers between requests while keeping the same user credentials. We deploy automated browser agents to send these requests and record the responses in a central database. Finally, we review and analyze flagged requests to manually confirm vulnerabilities.

6.1. Semi-Automatic User Accounts Management

We integrate the *Account Framework* [21] to store and manage the user accounts for the various websites. As a first step, we perform manual registration through the provided assisted setup to create two user accounts per website keeping track of e-mails, usernames, and passwords. All users share one of three e-mails we control (one Gmail account and two Proton [34] accounts). Rautenstrauch et al. [21] used the created accounts in a fully automated fashion by crawling the sites through the following links. They did not, however, interact with the page further (e.g., triggering forms or buttons). In contrast, we must perform certain unskippable actions (e.g., filling out the profile information on the first login) in *both* accounts. For this, we develop a protocol to ensure that our two accounts are in the same state before we start interaction with the *leader* browser.

The *Account Framework* then automates the rest of the process to serve fresh sessions for the Variable Swapping Framework (Figure 1), by automating any required logins for expired sessions. We refer the reader to their paper for a detailed explanation of the inner workings of their setup.

6.2. Manual Mirrored Website Visit

For each endpoint to evaluate *AC* behavior, we need examples of accepted resource identifier values for both users (e.g., folder ID), so that we can later swap between users (Section 6.4) to ensure only researcher-controlled is accessed. Effectively, we must collect an example request for the site endpoint from both user sessions. Naturally, we cannot claim to exhaustively evaluate all endpoints for a given site; we rather restrict ourselves to covering the *same* set of endpoints between two sessions to have the necessary example requests in both cases.

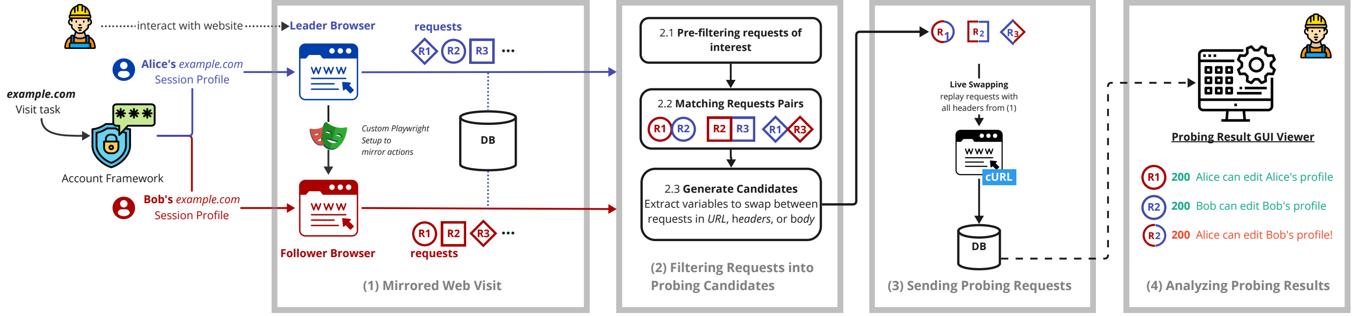

Figure 1. The Variable Swapping Framework (VSF) Simplified Workflow

To efficiently explore available endpoints and collect two examples from two distinct user sessions, we design a mirrored browser setup, as shown in Figure 1 (1), maneuverable by a single researcher. The researcher controls one browser, labeled *leader*, and navigates the website with the first user account (Alice). A second browser, labeled *follower*, automatically mirrors each interaction (clicking, filling inputs, navigating URLs) produced by the *leader*, but with the second user account logged in (Bob). Intuitively, synchronizing the browsers' interactions guarantees a higher likelihood of generating two examples of each discovered endpoint from the two user accounts. We collect request/response pairs from each browser in a database for later stages of the experiment. In case the follower browser falls out of sync, the researcher can directly interact with it to bring it back to the same state as the leader browser. We elaborate more on the implementation details of this setup in Appendix A.1.

6.3. Filtering Requests into Probing Candidates

This step's goal is to minimize the number of irrelevant requests and useless swapping experiments reducing the analysis workload and the network impact on the tested server. We disqualify requests and probing candidates in three stages, visible in Figure 1 (2):

I. Pre-filtering Requests. Since we focus on authorization issues, we can eliminate many requests that do not involve any *AC* pattern. First, we drop all requests to *third-party* domains that match popular Ad-blocker list, EasyList [35], as they are likely *Advertisement and Tracking Services (ATS)* which are not relevant to our study – using `braveblock` Python library [36]. Next, we only include requests to resources that may contain sensitive information, such as `XHR`, `HTML`, `XML`, `TEXT`, `JSON` and `IMAGE`. In turn, we excluded mostly likely static content like CSS. Then, we design a Regex filter list to only allow requests with any form of authentication (e.g., cookies, authentication header, authentication keyword in the request body, etc.). Finally, we drop identical request/response pairs in both sessions, as surely, they do not include any user-specific credentials.

II. Matching Requests from Visit Pairs. Multiple factors can cause request sequences to be misaligned between browser sessions: JavaScript randomness can cause certain requests only in one of the sessions, and timing differences can cause requests to be in a different order, randomness in server response between users can trigger slightly different control flows, etc. We cannot simply assume that the ith request from Alice's session and the ith request from Bob's session are examples of the same endpoint. To match request pairs, we design a heuristic that computes the distance between requests of two distinct browser sessions based on the URL path similarity. We compute URL path similarity between URL A (e.g., `/api/profile/alice`) and URL B (e.g., `/api/profile/bob`) by counting the number of matching segments at the same index, $\text{sim}(A, B) = \sum_i \mathbf{I}_{A_i = B_i}$ where $\mathbf{I}$ is the indicator function. Given two request sequences $\mathbb{R}^{\mathbb{A}} = \{R_i^A\}$ and $\mathbb{R}^{\mathbb{B}} = \{R_i^B\}$, We consider the matching request from $\mathbb{R}^{\mathbb{B}}$ for R_i^A to be the request having the smallest distance with it:

$$\text{match}(R_i^A, \mathbb{R}^{\mathbb{B}}) = \arg\max_j \text{sim}(R_i^A, R_j^B)$$

If the shortest distance is larger than a fixed threshold T_R, the framework disregards the pair, otherwise, the pair is considered a viable probing candidate and is handed over to "candidate generation". We experimentally determine that $T_R = 3$ yields the best matching results by manually tuning it on requests from a pilot experiment.

III. Probing Candidate Generation. We aim to generate a request to an endpoint a that takes credentials from one user and the resource identifiers from the others to detect a potential *AC* flaw. We refer to the user providing the credentials as U_c: the associated *AC* rule is (u_c, r_c, a), where r_c is the resource identifier for this user, and the associated request-response pair is R_c. Similarly, we refer to the user providing the resource identifier as U_r with associated *AC* rule (u_r, r_r, a) and request-response pair R_r.

First, we design a request templating process (Figure 2): (1) Given the request information from R_c (URL path, GET query parameters, headers, and POST body if any) and a set of variable values present in the request (e.g., folder identifiers), we can construct a "request template" where each instance of the parameter value is replaced with a regex

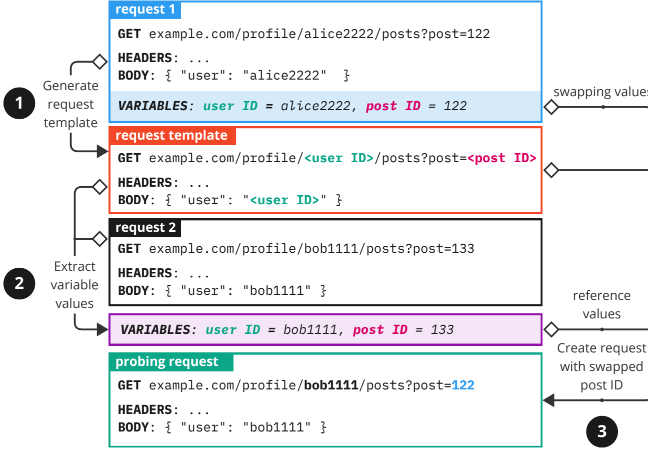

Figure 2. Example of generating a probing request from visit request pairs

pattern; (2) Using this template, we can extract the values for the same set of variables for the other user's request R_r; Finally, (3) knowing both sets of variable values for the variables, we can plug them back into the request template to generate requests R_s with swapped resource parameters for disallowed AC examples (u_c, r_r, a). You can find more about how we compare requests in Appendix A.2.

Identifying request variables. Automating the templating process still requires enumerating the list of variables from each request. One option is to note them manually, which we offer on the framework through an interface for advanced uses. However, this approach does not scale, requiring additional manual labor. Thus, we design an automated heuristic approach to identify parameters worth swapping (i.e., likely AC parameters) according to the parameter keys and value formats. For example, UUID v4, which could be relevant AC parameter, has distinct value formats `xxxxxxxx-xxxx-4xxx-yxxx-xxxxxxxxxxxx` where `x` is a hexadecimal digit and resource identifiers often include `*_id` in their name. We maintain allowlists and blocklists of parameter names and values which we curate manually from a pilot experiment. You can find more details about the heuristic patterns in Appendix section A.3.

Finally, we can generate multiple probing requests if the reference request has many resource identifiers. From the set V of available identifier, the candidate generation module selects a subset of variables V_S to swap based on the heuristic described previously. Then, it can generate all $2^{|V_S|}$ combinations. To avoid overwhelming the server and keep the time-spent per site short, we put a hard limit of 16 candidates per endpoint. In our experiments, we found that this limit was rarely reached (only 3% of probing requests). This limit can be easily adjusted in the future to fit the needs of the study.

Heuristic Evaluation. We evaluate the performance of the filtering module over 3,000 request-pairs randomly selected from all collected request-pairs (16%). We manually go through each pair and label whether it contains swappable resource identifiers and would be a valid probing request or

Table 2. CONFUSION MATRIX FOR THE PREDICTIONS OF THE PROBING CANDIDATE FILTERING HEURISTIC.

	True YES	**True NO**
Predicted YES	203	297
Predicted NO	83	2,390

***YES**: the request-pair is AC-related and should be probed, **NO**: request-pair not interesting to probe for.*

not. In Table 2, we show the confusion matrix between the labels predicted by the filtering heuristic and those manually labeled. As mentioned in Section 7, the heuristic favors false positives to try to cover as many examples as possible, 297 candidates or an $FPR = 59\%$, as we provide relaxed patterns. However, we can see that the number of false negatives is comparably smaller, with only 83 or an $FNR = 3.3\%$. By extension, the heuristic achieves a precision of 40% and a recall of 71%. The regex patterns used for the heuristic can be easily updated to better fit the examples of future studies.

6.4. Sending Probing Requests

The key requirement of the swapping module is ensuring that the difference in server-side responses to probing requests compared to visitation requests is only due to the swapped parameters. The main factor that can cause the server to respond differently besides the swapped parameters is expired security parameters: Authentication credentials (e.g., session cookies, JWT tokens, etc.) can expire if the swapping experiment is sufficiently delayed, causing the server to respond with an unusable authentication error response. We guarantee this *parameter freshness*, by running a live swapping experiment in parallel to the visitation. On a rolling basis, the swapping worker fetches probing requests generated by the prior module (Section 6.3). This approach guarantees that short-lived session parameters – like CSRF tokens – remain valid and do not influence the response to probe requests. Still, live swapping can influence the visitation session if the requests succeed (e.g., deleting a user folder), which may be confusing to the researcher running the experiment. In our measurement, though, no such situation arose. We intentionally send all of the probing requests outside of a real browser with cURL [37], to allow us to send arbitrary headers (which would cause CORS-related preflight requests if done from within page JavaScript).

6.5. Manual Candidate Validation

We use the GUI viewer to understand whether (1) successful probing requests (with status code 200) present actionable vulnerabilities and (2) failed probing requests failed due to AC policy violation or orthogonal reasons.

Ideally, we expect the response R_s to the swapped AC rule (u_c, r_r, a) to return an HTTP error code. As such, successful responses to probing requests are markers of likely vulnerabilities. To validate whether such candidates violate the expected AC policy, we perform a series of

manual checks: (1) we contextually determine whether the swapped parameter ($r_c \leftrightarrow r_r$) is a private resource identifier. If yes, (2) we compare response bodies: if R_s is closer to R_r (which shares the same resource identifier) than R_c (which shares the same credential), we consider it a **violation of the *AC* policy**, as user u_c was able to access u_r's resource r_r. In case no response body is available (for `POST` requests for example), we consider successful requests over private resource identifiers a **violation of the *AC* policy**. While this can be automated [7], we manually confirm that the improper authorization is an exploitable vulnerability, so that we do not notify operators incorrectly.

7. Evaluation

We use the framework described in Section 6 to scan web-app endpoints for *AC* vulnerabilities and explore the reactions of servers to mismatching *AC* requests. In the following sections, we present the results of our experiment from selecting the websites that we can test, to performing the visits and collecting requests, and finally generating and triggering probing requests that uncover serious vulnerabilities. In Table 3, we provide an overview of the relevant counts of websites and requests throughout the various stages of website selection and the VSF pipeline. In it, we can see the reduction in the number of involved websites and requests as they become subject to the technical and ethical constraints we impose and the scope we aim for. This distillation of websites and requests through the automated process helps reduce the number of endpoints to investigate from 60K endpoints collected in Section 6.2 to 584 in Section 7.3.

7.1. Website Selection

While all other stages scale to any number of websites, we had to constrain the number of visited sites due to the manual load to register accounts and visit websites. We still attempt to cover sites relevant to users by sampling from the CrUX dataset [38]. The Account Framework [21] detected 531 websites across the 0-5K range as having login and registration pages, of which we could create account pairs to 110 websites between Aug. 2024 and Oct. 2024.

7.2. Testing Web Endpoints in the Wild

This section showcases the results obtained from visiting live websites with the VSF. To demonstrate the design and utility of our framework, we decided to only apply it to 100 sites on which we could establish two sessions. As we discuss in the following section, 10 websites failed for various reasons, hence we registered accounts with a total of 110 sites for experiments.

Visiting Websites and Interactions. We intentionally do not attempt to automate website visitation and interaction. This manual step ensures unintended unethical interactions

Table 3. GENERAL EXPERIMENT STATISTICS FROM VISITATION TO PROBING REQUESTS

Website Statistics	
Websites with registration and login forms	531
Websites with two accounts	110
Successfully visited websites	100
Websites with probing candidates	90
Visitation Request Statistics	
Total requests collected	536,973
Unique endpoints collected	60,153 (11.2%)
First-party requests	24,352 (40.4%)
Third-party requests	35,801 (59.6%)
Unique requests dropped by ad blocker	27,609 (45.9%)
Pre-filtered and matched endpoints	7,311 (12.2%)
Probing Request Statistics	
Unique request to swap	1,566 (2.6%)
Total probing candidates	6,830
Successful probing candidates	6,096
Unique probing URL templates	584
Endpoints improperly handling disallowed *AC*	30
Vulnerabilities	19
Vulnerability Breakdown By Adversary (Section 4)	
Opportunistic brute-force attacker	6
Opportunistic reconnaissance attacker	13
Targeted attacker	6

cannot occur (e.g., interacting with a real user). It would be challenging to give ethical guarantees without tightly controlling the interaction footprint of our users by hand or sacrificing interactions altogether like prior attempts [7].

Visitation Guidelines. For each website, we enforce a clear visitation protocol. In essence, we instruct the researchers visiting the website to perform interactions that would *(A) expose AC-related endpoints* (e.g., navigating/editing the user profile, manipulating user-owned objects, liking/bookmarking public resources, etc.) without *(B) compromising the ethical bounds* (e.g., not reporting other users). For a full protocol description, we refer the reader to Appendix B. All subsequent stages are automated and ensure that parameters making it to the probing requests are only pertinent to one of our two users, not harming any third-party user.

Failed Visits. Out of the 110 websites we visited, we marked 100 as successful. Out of the 10 failed visits, three websites, marked as logged in by the Account Framework (Section 6.1), would not provide logged-in sessions during visitation. Further investigation shows our accounts were either blocked and required an account recovery process or bot detection invalidated our sessions. Three other websites failed because we could not have any meaningful interaction. We further outline interaction limitations in Section 8.2. The remaining websites failed because the web pages were too dissimilar between users or very unstable. The three main reasons for session divergence between users are (1) AB-testing where a different user interface is selectively shown to one portion of the users e.g., how the share page responds on `notion.com`, (2) randomized content in the landing page e.g., images shown on `pinterest.com`, and (3) dif-

ference in user account states e.g., one user getting a random achievement on `boardgamearena.com`. While the VSF falls back to clicking in the same position of the page if the interaction target is missing from the *follower* browser, differences in UI layouts will quickly lead both browser sessions to become out of sync. Finally, we note some websites, primarily adult content websites, that implement aggressive advertisement techniques by opening many new tabs at every click, which renders a coherent visit infeasible.

Website and Interaction Diversity. For scalability, our framework should handle a wide range of website categories and interaction patterns. After visiting a large portion of websites, we identified a finite number of popular high-level interaction sequences that we adopted: (*A1*) editing the profile page information of the user; (*A2*) navigating all private sub-pages, e.g., the user's favorite videos list; (*A3*) opening the notifications list; (*A4*) interacting with any messaging service *only between both our users*, e.g., opening inbox, sending a message, etc.; (*A5*) creating/updating/deleting user-owned objects, e.g., adding a playlist, renaming it, or deleting it; (*A6*) making associations with public objects, e.g., liking an image, bookmarking an article, adding a video to our playlist, temporarily following a user etc., (we avoid any action that harms other users); (*A7*) interacting with 3rd-party services such as payment services reversibly, e.g., we enter the checkout page but never proceed to payment. These interaction classes allow us to group websites into distinct categories, which can be useful for future attempts at automated interactions: 41 **media gallery** web sites, e.g., video pages, adult pages, image galleries etc., focusing primarily on user-gallery interactions *[A1,A2,A6]*; 13 **news or articles** web sites, where *AC*-relevant interactions are minimal *[A1,A6]*; 12 **social network** web sites offering the largest diversity of interactions including user-to-user interactions *[A1,A2,A3,A4,A5,A6]*; 12 **product gallery** web sites limited to user-to-product interactions and cart services *[A1,A6,A7]*; 11 **content editor** web sites, e.g., PDF editors `ilovepdf.com`, image conversion `cutout.pro`, course-taking `w3school.com` etc., where users often upload private files and/or manipulated objects private to them *[A1,A2,A3,A5]*; seven **information retrieval** web sites, e.g., search indices like `britannica.com`, with rare meaningful *AC* interactions *[A6]*; lastly four **file manager** web sites, e.g., `mediafire.com`, `github.com`, etc., with many high-impact *AC* interactions *[A1,A2,A5]*. Since the VSF mirrors interactions at the atomic level of clicks and form inputs, we can see that it can handle most of the web easily.

Uncovering Testable Site Endpoints. Given the black-box nature of the experiment, a prerequisite for the primary goal of the VSF is triggering and exploring as many site endpoints as possible, collecting two occurrences of the endpoint from both user sessions, to use in the swapping experiment.

First, to argue about the likelihood of having two examples of any request, we need to measure the overlap between requests across the pair of visits to the website. Removing all requests blocked by an ad-blocker (Section 6.3), we take

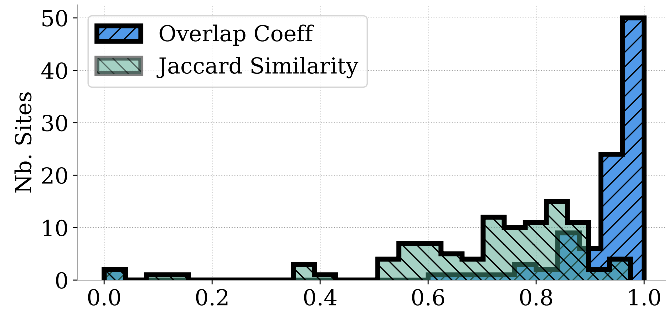

Figure 3. Request overlap and Jaccard similarity of requests observed in mirrored visit pairs to the same website.

the pairs of request URL sets $\mathbb{R}_A$ and $\mathbb{R}_B$ and compute the *overlap coefficient* and *jaccard similarity*:

$$\text{overlap}: \frac{|\mathbb{R}_A \cap \mathbb{R}_B|}{\min(|\mathbb{R}_A|, |\mathbb{R}_B|)} \qquad \text{Jaccard}: \frac{|\mathbb{R}_A \cap \mathbb{R}_B|}{|\mathbb{R}_A \cup \mathbb{R}_B|}$$

Our results in Figure 3 show that 75% of visit pairs have a Jaccard similarity higher than 61% and more than 90% overlap. The high overlap confirms the success of the mirrored browser setup. The larger spread in Jaccard similarity compared to the concentrated high overlap indicates that one of the two visits observes more varied requests than the other. We attribute this to two reasons: (1) in some cases, we need to perform interactions only on the *follower* browser if the browsers lose synchronization, and (2) the *leader* browser closes slightly before the *follower* which causes more interactions to be recorded on the follower.

Second, in our threat model (Section 4), we expect adversaries who only have black-box access to the server, to register as a user, and explore vulnerable site endpoints by interacting with the website in manners similar to ours. As such, we need to characterize the nature of the requests we encounter using the VSF to illustrate the advantage of the adversary and the coverage of our method. Due to our ethical constraints (Section 5), we refrain from comparing to other web-testing tools on production websites [5, 6, 7, 8] and include a static crawler (a crawler that only navigates to sub-pages from page links without interacting with the content) as a baseline, by estimating its performance from portions of our visits before any interaction. This estimation is an upper bound on the performance of static crawlers, as we must include pages that result from user actions – unreachable to a typical static crawler. We only perform this analysis over one of the browser sessions per visit. Our results in Figure 4 show that interactive crawls cover three times more unique requests (31,457 request) and two times more unique domains (2,380 domain). We expect most of the relevant endpoints to be first-party requests (Section 6.3 II). The breakdown also shows that the interactive visits with the VSF lead to a median of 262 first-party request per subject compared to only 72 with a static crawler. Overall, we cover 24,352 unique first-party requests from all sessions and websites. After the pre-filtering and matching process, we end up with 1,566 request pairs of

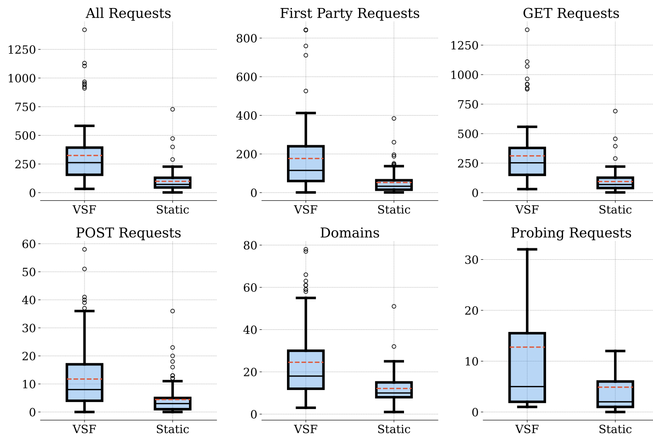

Figure 4. Comparison between interactive mirrored website visitation (VSF) and crawling it statically (Static) regarding the count distribution of unique requests and domains encountered per visit.

which 970 are unique; resulting in 2.6 times more requests compared to static crawling. These are the requests that the VSF considers *AC*-relevant and within our threat model. Candidate generation and templating (Section 6.3 III) further deduplicate distinct request URLs into 584 candidate URL templates, e.g., /users/alice1111/1234.json and /users/bob2222/5678.json are distinct but reduce to the same candidate URL template /users/<user ID>/<file ID>.json (Figure 2).

Interacting with the web app allows the client to reach deep application states not normally reachable to a static crawler. Changing application states normally occurs after POST requests (e.g., saving a profile, creating a post, etc.). We also verify that interactions allow us to reach intermediary application states and send POST requests normally unreachable 1,140 unique POST requests, 2.6 times more than a static crawler, of which 18% become probing candidates. On top, we also generate 30,236 unique GET requests 3.3 times more than static crawlers. Note that the higher prevalence of GET requests is expected since it includes (1) any request containing static resources which often contain randomized parameters inflating their numbers, e.g., example.com/_static/123456-789.jpg and (2) advertisement and tracking services requests.

Overall, our VSF visitation experiments reveal a diverse set of endpoints and more chances to detect *AC* issues from the black-box servers than traditional crawling.

Manual Candidate Validation. We design our filtering strategy to be more forgiving of false positives, as (1) we expect *AC* vulnerabilities to not be very abundant, (2) combing through the candidates using the GUI we designed makes it easy to filter out emerging patterns of false positives on the go, and (3) it is easier to drop candidates we do not find useful than to start a new swapping experiment for every new candidate skipped in prior iterations on the VSF. Recall that we generate many probing candidates per endpoint, one for each combination of variables we choose to swap or keep as the reference, Section 6.3 III: the 584 templates result in 6,096 probing requests. We manually go through all groups of probing requests having the same URL template, and choose one or more representatives to analyze manually depending on the variety of response codes and the importance of the value swapped in the website's context, e.g., for /api/<user ID>/<file ID1>?backref=<file ID2> we likely care more about file ID1 as it is the file the user will be visiting. For practitioners deploying our tool, manual analysis is only required for probing responses with "unexpected" status codes, i.e., 20X or 30X codes. On average, a practitioner to manually analyze 6 requests per site on average. On top, 43% of such requests correspond to public or static resources that escape the endpoint filtering (Section 6.3). Fine-tuning our filtering heuristics can further reduce the manual analysis overhead. The remaining 3 requests per site on average require in-depth *contextual* analysis to understand harm and exploitability.

As expected, only 221 request templates (38%) pertain to *service endpoints* for the main site functionality. The largest source of false positives are endpoints interacting with public resources with 254 (43%) URL templates of which 213 represent static resources. The VSF attempts to manipulate these requests because they often contain many differentiated identifier-like segments in the URL. Such resources can be easily filtered out by including them in the filter lists for the filtering module, e.g., URLs including _static, assets, etc.. The remaining 41 public templates are service endpoints: one such example is endpoints to bookmark an article /api/bookmark?article_id=1. Such requests contain all the elements we are looking for in an *AC*-related request: *credentials* (through request headers or session cookies), *resource* identifier (article_id), and an action (bookmark). It is only by contextually understanding that the object here is public that we rule it out. The second-largest source of false positives is analytics, logging, advertisement, and tracking endpoints, 110 URL templates (17%), which have a similar structure to service endpoints: a user identifier, an resource identifier, and an action e.g., (log_click). While we attempt to filter out tracking requests with an ad-blocker module (Section 6.3), first-party analytics, and logging requests are often allowed. Next, 40 templates attempt to swap non-*AC*-related parameters, often in the URL path, since such parameters do not have a name or key that the VSF can use to filter their purpose. Finally, the remainder of the unfruitful templates try to swap parameters in security requests to, e.g., Cloudflare and CloudFront.

***AC* Patterns.** This section showcases the main patterns we observed in the sites' approaches to *AC*. First, we notice that most endpoints (73.6%) rely on cookie-based credentials, e.g., a shared session ID or authentication tokens. Four times less frequently (18.4%) are authentication tokens implemented in the headers directly, e.g., JSON Web Tokens (JWT), client key, etc.. We attribute this to legacy practices and the ease of storing and transmitting cookies – since the browser handles it automatically. In comparison, developers have to handle the storage and transmission of authentication

headers themselves. Lastly, 13 endpoint templates embed credentials in the request query or body; such requests are usually third-party API calls, where the credential is a separate client key. As for resource identifiers, most request templates (53.9%) embed them directly in the URL segments, e.g., `/api/folders/<folder ID>/files`, followed by query parameters (26%). This is prevalent in `GET` requests, especially navigation requests, to impose structure and consistency in the page or API schemes. However, such parameters are more accessible to attackers and the public, visibly in the URL bar, and easily manipulated by a motivated attacker. More complex endpoints (18%) use the request body and the `POST` method, especially if they are handling sensitive operations like a shop checkout. Lastly, two templates use cookies to store references to resource identifiers, namely for maintaining a stateful user cart in an online shopping store.

Second, we noticed multiple cases where redundant user identifiers were present in the same request, e.g., user ID, email, and username. As we show in Section 7.3, such practices can cause serious vulnerabilities. Causing a similar worry, we found many resources to have global identifiers despite being wholly private to users, e.g., on RedactedEditor.com any document uploaded is referred to with a global long identifier, even through shareable links.

Finally, we evaluate the error handling of the *AC* policies and their impact on our analysis. We identify four main trends in handling unauthorized requests without causing vulnerabilities: (1) 66 templates throw a 401, 403, or 404 error code to signal that the resource requested is out of bounds for the user; this is the most appropriate response, but it can come with privacy concerns if the server leaks too much information about the error (e.g., "You cannot access this. This item is owned by user Y" on RedactedPublisher.com). (2) Four templates return a 200 code response but contain a custom error in the response body; such approaches are not necessarily unsafe, but they make it harder to automatically find vulnerable endpoints. (3) Four templates return a 200-code response and attempt to remedy the situation by suggesting a redirect link, showing a public version of the page requested, or ignoring the parameter and serving the response appropriate to the credential; such approaches are not recommended as they can easily malfunction or trick users, e.g., if user Y requests `/profile?username=X`, it can show the profile data of Y but with the username of X. Finally, (4) three templates attempt an immediate redirect (30X response code) to the login page or an appropriate index page; we also find such responses suitable and simple.

7.3. Authorization Vulnerabilities

After the in-depth manual analysis (described in Section 6.5), we end up with 30 (5.5%) endpoints that improperly respond to requests with bad *AC* pairs across 15 sites out of 100. While improper responses are not necessarily immediate vulnerabilities, they can be a strong indicator that *AC* checks was not properly implemented, which developers could unknowingly transfer to other (more sensitive) parts of the application. Our exploit analysis concludes that 19 out of 30 improper responses are exploitable vulnerabilities with tangible harms: (H1) leaking private user information, (H2) partial credential hijacking, and (H3) user resource manipulations. The remainder of broken *AC* responses either (a) return successful responses but the responses themselves are not valuable (e.g., returning post recommendations) or (b) have resource identifiers that are ephemeral or impossible to leak (e.g., image hash for Amazon storage services). We enumerate here the main (non-exclusive) causes certain *AC* responses qualify as vulnerabilities: (R1) using identifiers with low entropy (e.g., incrementable, timestamp and incrementable, etc.) – such identifiers can be easily manipulated by the *opportunistic brute-force attacker* (5 vulnerabilities); (R2) supplying identifiers that can be inferred from the user credentials in a separate field, e.g., email in request body (1 vulnerability); (R3) using the same resource identifier for private endpoints and in endpoints accessible to other users or the public (e.g., URL leaks [28]) (13 vulnerabilities); and (R4) high entropy resource identifiers without any *AC* checks (6 vulnerabilities). We would like to note that 13 out of 19 vulnerabilities are exploitable by the weaker *opportunistic brute-force attacker* and *reconnaissance attacker* models, while only the remaining 6 require a *targeted attacker* model (Table 3).

In the following, we describe three serious vulnerabilities we found as examples of instances of flaws. At the time of this writing, we have disclosed the issues to vendors, but have not received confirmation of fixes nor explicit consent to name them, hence we redacted some details. A fourth example is available in Appendix C.

Vulnerability 1: RedactedNews.com . The first service we examine is a news website that allows users to make donations on one of its pages. This page initiates a request to */api/payment-provider/user*, containing the user's mail in the body JSON. In response, the server returns an access token for a payment provider which can be used to receive the user's payment information and purchase history. However, anyone can input any user's email address to obtain the other user's access token, thereby gaining unauthorized access to that user's payment information.

This unauthorized access could be exploited by a *targeted attacker* who first needs to know a valid user email address. However, since it is also possible to brute-force common email addresses or use leaked email lists, a weaker attacker model could also apply here.

To address this issue, we recommended that the vendor validate the authorization via the session user's email instead of relying on the JSON body input. Any input that could be manipulated by an attacker must be handled with caution.

Vulnerability 2: RedactedSocial.com . The next service is an adult dating website. Every profile can be previewed as a profile summary with the information received via the API endpoint */coreapi/profile_summary?handle=<user handle>&id=<profile id>*. This can be done using either a user handle or their user ID. While the handle is visible on

the website, the IDs are unique numeric identifiers used only by the API. The ID consists of two parts, totaling 12 digits, and appears only partially random. Retrieving someone's profile summary remains possible even if the corresponding user profile was set to invisible for other users.

An *opportunistic brute-force attacker* could easily enumerate the IDs to gather user information about several users who intended to be hidden. A *targeted attacker* could achieve the same by directly using a specific user handle.

This issue originates from the API's failure to verify whether a profile is set to invisible or not. We, therefore, recommended the vendor to implement this check within the API to resolve the issue.

***Vulnerability 3: RedactedEditor.com* .** The third vulnerability we describe was found in a PDF signature service. A user can upload a PDF and distribute the document to other signers via mail. For a new signature task, a task token is created. Visiting `/signature/request/<token>`, the user can see and configure the signature request. Using the same token with the API `/v1/signature/user/<token>` further reveals information about the signers and files. Although the signature task is bound to accounts via email addresses, every user possessing the token can open the request, download the linked files, and view all signers' info.

The token itself is 70 random characters long so presumably very secure. Since the signature requests are sent via e-mail (which could potentially leak the token), this is however a prime example of a vulnerability that falls in the *opportunistic reconnaissance attacker model*.

To resolve this vulnerability, we would recommend verifying the request's authorization by checking both the session user's email and the signers' emails. Relying only on a randomly generated token in the URL is not enough.

These three vulnerabilities were discovered due to unique design features of the VSF: First, vulnerabilities (1) and (3) require sending `POST` requests, which prior work avoids due to ethical concerns; the VSF can send `POST` requests because we are sure to manipulate our own users' data only. Second, vulnerability (3) requires a complex setup of specific resources and user states (e.g., creating a signature request), which is only possible ethically with controlled manual visitation in the VSF.

8. Discussion

Our results show that the Variable Swapping Framework is capable of detecting critical authorization issues efficiently, mainly thanks to the live browser mirroring. On one side, it allows the practitioner to explore a diverse set of interactions and utilize their domain knowledge to poke more on the sections of the site where more sensitive processes take place, e.g., cart checkout, messaging a user, etc.. On average, visitation takes 7.75 minutes (σ = 3.9) per site. On the other side, the almost-perfect synchronization (Figure 3) of the visits between users ensures that we can almost always find two examples from an endpoint to generate probing candidates and evaluate the *AC* of it. The modular and upgradable filtering heuristics reduce the set of candidates to probe and to be later analyzed manually. We show in Section 7.2, only 6 requests per site on average need manual validation, only 3 of which require deep vulnerability analysis. Automating deep vulnerability assessment is a challenging problem, as it requires understanding the semantics of the response, the context of the request, and the user's state; we regard this as distinct topic for future work. We discuss more about the usability and scalability of the VSF in Appendix D.

With the VSF, the only manual bottleneck is the site visitation. In reality, the VSF is agnostic of the driver of the visitation, be it a human or an automated agent. However, guaranteeing an ethical automated visit is presently challenging, due to the many unethical interactions that look very similar to ethical action, e.g., messaging `bob1` who is a third-party user instead of `bob2` who is our controlled user, reporting a user instead of following them, etc. Evaluating interactive agents on their ethical grounding is an interesting problem for future work.

Finally, regarding the access control in the wild, we find that 15% of the sites incorrectly handling bad *AC* requests are not negligible. This confirms the high rank of the access-control vulnerabilities in OWASP's Top 10 [1]. Most issues occur because of redefining the identity through variables other than the credentials, e.g., having a separate `username` field, or because of relaxing the need to maintain *AC* on resources referenced by long and complex identifiers. Even when *AC* responds correctly, the variance in how it reports forbidden access varies widely between sites, making it harder to automate the analysis, e.g., returning a 200 code response but containing an error JSON object in the body.

8.1. Responsible Disclosure

We reported all 19 vulnerabilities to the corresponding parties using the most appropriate security channel we could find, including bug bounty programs, privacy email addresses, and in many cases generic email addresses such as `contact@` or `help@`. At the time of writing, we did receive feedback from several operators with one providing a bounty payout, while others acknowledged and accepted the reports. None of the responses denied the issues raised.

8.2. Limitations & Future Work

As with any empirical tool, we encountered many limitations and constraints that shaped the design of the VSF and our methodology. We briefly enumerate in the following sections the main sources of limitations. We also discuss avenues to overcome some limitations in future work.

Unexplored Interactions. While visiting websites, we abstain from many interactions for ethical and logistical reasons: (1) we do not perform any interactions requiring payment; this includes unlocking website sections available to paid subscriptions, proceeding with a purchase from an online shop etc.. (2) we consider it unethical to interact with

third-party users directly as it would constitute spam activity; this includes sending them messages, reporting them, etc.. We still perform passive actions like following them, liking their posts etc. as long as we reverse the actions afterward. (3) we do not permanently influence public resources, e.g., submitting a form to contact IT support or rating a product without removing the rating immediately afterward. (4) we do not update username or password sections of the user profile, as it might log out our users, and compromise the mirrored visitation session. Finally, (5) we ignore actions that require a complex user state, e.g., only users who rated five products on a store can add a profile picture.

Different User Roles. The current configuration of the VSF supports only users of the same role. However, several parts can be extended to support different roles, e.g., by using the parameter extraction heuristic (Section 6.3) to build global variable values dictionaries that users of different roles can share across distinct API requests. The mirrored visitation is less useful in cases where users of different roles have non-overlapping website layouts. We do not explore this in the current work, as most websites we evaluated only expose a single role to the user. Generating higher-privilege users would require provider-cooperation, which is out of the scope of this black-box work.

Website Limitations. Some websites are not easy to visit and this can happen due to many reasons: Site language that might be hard to parse by certain researchers; sites that try to load large content such as high-quality videos or custom animations that cause a considerable lag; And finally, sites with many redirects and randomly opening new tabs – this often happens due to advertisements.

8.3. Recommendations

As we believe we have demonstrated with this work, *AC* research in a real-world environment is ethically possible if ethics are considered for every step of the research process. For researchers who aim to focus on the server-side issues in the real world, we recommend following a similar approach and also conducting a stakeholder analysis as suggested by previous ethics work [32, 31] and as outlined in Section 5. This exercise encourages the researcher to deeply assess the potential risks and benefits their work may present to individuals affected. The literature in the past years started to provide useful guidance on ethical practices and potential pitfalls, with work continuing to evolve in this field [32, 31, 9, 39]. In general, we believe that this exercise is a good practice for any study to understand their place in the context of society, fostering more responsible and refined research. We also provide recommendations for developers based on the insights from vulnerabilities in Section E.

9. Conclusion

Being unable to study server-side flaws such as improper access control in the wild comes with significant detriments for the security of the web ecosystem. In this work, we show that such studies are feasible in an ethically sound fashion by carefully considering all stakeholders and designing a measurement framework to abide by a strict set of rules while finding (and disclosing) real-world flaws.

Our extensive stakeholder (Section 5) analysis highlights that the VSF design addresses the largest ethical questions the experts raised: Our mirrored browser setup with two proprietary accounts (Section 6.2) prevents cases where we accidentally target a third-party user without consent. Our strict visitation protocol and limits on interactions ensure we cannot harm third-party users, e.g., reporting their accounts, messaging them, etc.. Finally, limiting the rate and number of probing requests ensures we do not abuse server resources.

Second, we study the utility of the VSF in finding improper *AC* patterns and vulnerabilities in the wild. We visited 110 websites and generated 6,069 probing requests for 584 unique endpoints. Our results show that the mirrored visits maximize the number of testable endpoints by providing example requests to these endpoints from both users. Requests are compared live and filtered to include only *AC*-related requests while allowing for false positives to maximize exploration. Our manual validation of the probing requests uncovers 30 improper *AC* endpoints across 15 websites. As we find that 19 of these endpoints can be exploited as vulnerabilities, we conclude that the VSF reached its intended purpose without crossing ethical lines.

We deliver this framework as proof that server-side scanning on live websites is possible without compromising legal and ethical limits highlighted by Hantke et al. [9]. We encourage future work to let ethical requirements shape their frameworks to maximize the utility of the study and minimize the harm to the stakeholders involved in the experiment.

References

[1] OWASP. (2021) OWASP Top 10. [Online]. Available: https://owasp.org/Top10/

[2] B. Krebs. (2019, May) First American Financial Corp. Leaked Hundreds of Millions of Title Insurance Records. [Online]. Available: https://krebsonsecurity.com/2019/05/first-american-financial-corp-leaked-hundreds-of-millions-of-title-insurance-records/

[3] HackerOne, “8th annual hacker-powered security report,” 2024.

[4] F. Liu, Y. Shi, Y. Zhang, G. Yang, E. Li, and M. Yang, “MOCGuard: Automatically Detecting Missing-Owner-Check Vulnerabilities in Java Web Applications,” in *IEEE SP*, 2025.

[5] Y. Ishida, M. Hanada, A. Waseda, and M. W. Kim, “Automated vulnerability assessment approach for web api that considers requests and responses,” in *IEEE ICACT*, 2024.

[6] N. B. Chaleshtari, F. Pastore, A. Goknil, and L. C. Briand, “Metamorphic testing for web system security,” *IEEE Transactions on Software Engineering*, 2023.

[7] M. Rennhard, M. Kushnir, O. Favre, D. Esposito, and V. Zahnd, “Automating the Detection of Access Control Vulnerabilities in Web Applications,” *SN Computer Science*, 2022.

[8] G. Deepa, P. S. Thilagam, A. Praseed, and A. R. Pais, “Detlogic: A black-box approach for detecting logic vulnerabili-

ties in web applications," *Journal of Network and Computer Applications*, 2018.
[9] F. Hantke, S. Roth, R. Mrowczynski, C. Utz, and B. Stock, "Where are the red lines? towards ethical server-side scans in security and privacy research," in *IEEE SP*, 2024.
[10] S. Calzavara, R. Focardi, M. Squarcina, and M. Tempesta, "Surviving the web: A journey into web session security," *WWW*, 2018.
[11] M. Squarcina, P. Adão, L. Veronese, and M. Maffei, "Cookie Crumbles: Breaking and Fixing Web Session Integrity," in *USENIX Security*, 2023.
[12] S. Calzavara, H. Jonker, B. Krumnow, and A. Rabitti, "Measuring web session security at scale," *ACM CSEC*, 2021.
[13] I. Chenchev, A. Aleksieva-Petrova, and M. Petrov, "Authentication mechanisms and classification: a literature survey," in *Computing Conference*, 2021.
[14] J. S. Park and R. Sandhu, "Secure cookies on the web," *IEEE internet computing*, 2000.
[15] R. Kraft, "Research and design issues in access control for network services on the web." in *ICOMP*, 2002.
[16] J. M. Harán. (2019) A vulnerability in Instagram exposes personal information of users. WeLiveSecurity ESET. [Online]. Available: https://www.welivesecurity.com/2019/09/12/vulnerability-instagram-private-information/
[17] dakitu. Nord Security Disclosed on HackerOne: IDOR allow access to payments data of any user. HackerOne. [Online]. Available: https://hackerone.com/reports/751577
[18] datph4m. (2022) Tiktok Disclosed on HackerOne: IDOR delete any Tickets on ads.tiktok.com. HackerOne. [Online]. Available: https://hackerone.com/reports/1475520
[19] C. Brook. (2016) Uber Portal Leaked Names, Phone Numbers, Email Addresses, Unique Identifiers. Threatpost. [Online]. Available: https://threatpost.com/uber-portal-leaked-names-phone-numbers-email-addresses-unique-identifiers/122128/
[20] S. Roth, S. Calzavara, M. Wilhelm, A. Rabitti, and B. Stock, "The Security Lottery: Measuring Client-Side Web Security Inconsistencies," in *USENIX Security*, 2022.
[21] J. Rautenstrauch, M. Mitkov, T. Helbrecht, L. Hetterich, and B. Stock, "To auth or not to auth? a comparative analysis of the pre-and post-login security landscape," in *IEEE SP*, 2024.
[22] K. Drakonakis, S. Ioannidis, and J. Polakis, "The cookie hunter: Automated black-box auditing for web authentication and authorization flaws," in *ACM SIGSAC*, 2020.
[23] H. Jonker, S. Karsch, B. Krumnow, and M. Sleegers, "Shepherd: a generic approach to automating website login," *NDSS*, 2020.
[24] R. Kirchner, J. Möller, M. Musch, D. Klein, K. Rieck, and M. Johns, "Dancer in the Dark: Synthesizing and Evaluating Polyglots for Blind Cross-Site Scripting," in *USENIX Security*, 2024.
[25] CWE - CWE-639: Authorization Bypass Through User-Controlled Key (4.15). [Online]. Available: https://cwe.mitre.org/data/definitions/639.html
[26] CVE-2021-36539. CVE-2021-36539. [Online]. Available: https://www.cve.org/CVERecord?id=CVE-2021-36539
[27] NVD - CVE-2023-4836. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2023-4836
[28] C. Osborne. (2022) Urlscan.io api unwittingly leaks sensitive urls, data. PortSwigger. [Online]. Available: https://portswigger.net/daily-swig/urlscan-io-api-unwittingly-leaks-sensitive-urls-data
[29] R. Feng, Z. Yan, S. Peng, and Y. Zhang, "Automated detection of password leakage from public GitHub repositories," in *Proceedings of the 44th International Conference on Software Engineering*, ser. ACM ICSE, 2022.
[30] M. Ishida, K. Ikematsu, and Y. Igarashi, "Designing privacy-protecting system with visual masking based on investigation of privacy concerns in virtual screen sharing environments," *ACM HCI*, 2024.
[31] T. Kohno, Y. Acar, and W. Loh, "Ethical Frameworks and Computer Security Trolley Problems: Foundations for Conversations," in *USENIX Security*, 2023.
[32] M. Bailey, D. Dittrich, E. Kenneally, and D. Maughan, "The menlo report," *IEEE SP*, 2012.
[33] B. Stock, G. Pellegrino, C. Rossow, M. Johns, and M. Backes, "Hey, You Have a Problem: On the Feasibility of Large-Scale Web Vulnerability Notification," in *USENIX Security*, 2016.
[34] Proton mail. Proton AG. [Online]. Available: https://proton.me/mail
[35] Easylist. EasyList. [Online]. Available: https://easylist.to/
[36] Intsights/braveblock github. [Online]. Available: https://github.com/ArniDagur/python-adblock
[37] Curl. [Online]. Available: https://curl.se/
[38] Google. (2024) Chrome UX Report. Chrome Developers. [Online]. Available: https://developer.chrome.com/docs/crux/
[39] L. Bauer and G. Pellegrino. (2024) USENIX Security '25 Ethics Guidelines. USENIX Security. [Online]. Available: https://www.usenix.org/conference/usenixsecurity25/ethics-guidelines
[40] Microsoft. Playwright. [Online]. Available: https://playwright.dev/
[41] maggierui. (2024-10-03) How shareable links work in OneDrive and SharePoint in Microsoft 365 - SharePoint in Microsoft 365. [Online]. Available: https://learn.microsoft.com/en-us/sharepoint/shareable-links-anyone-specific-people-organization

Appendix A. Implementation Details

A.1. Mirrored Browser Setup

We use Playwright [40] to create the *leader* and *follower* Chrome browsers and load sessions fetched from the *Account Framework* (Section 6.1). To transmit interactions from the *leader* browser to the *follower* browser, we modify an existing Playwright feature: *Codegen*. *Codegen* is a mode to boot Playwright browsers into that records all user interactions and translates them into testing scripts to be replayed later. We modify the internal code of *Codegen* by attaching a WebSocket emitter for the *leader* browser that sends a WebSocket event for each Playwright action to a local signaling server we set up. The signaling server routes the events to the *follower* custom WebSocket client to execute them. Since executing scripted actions must ordinarily come from Playwright scripts, we could write the action into a script and then run it. However, we opted for a more elegant solution: First, note that Playwright allows for remote browser control, i.e., being able to run the browser on a different machine than the machine on which the Playwright process is running through WebSocket for performance reasons. So, we create a browser server for the *follower* and feed it fake instructions for each action from the signaling server as if coming from a playwright client process. Supported

Playwright interactions primarily include clicks and form-filling, which we find enough for most website interactions. To point the browser to the interaction element, Playwright uses a combination of XPath, CSS selectors, and text content search; but, we find that such locators can differ between the *leader* session and the *follower* session causing the *follower* not to find the element to interact with. An example we encountered was using the username as an element locator `p[innerText='alice']` which can only be found in the browser session logged in with Alice. To counter this, we modified Playwright to include a position-based locator as a fallback, i.e., tell the browser to click on the position (x, y) of the window. We found this workaround sufficient in most cases as the UI position is identical between browser sessions.

Minor Limitations. The codebase of the VSF is still in the early development stages, but we plan to work on it until we can release it as an open-source tool. For instance, so far only one tab of the browser is synchronized between users, and alert boxes are not synchronized. This means that we have to avoid actions that trigger new tabs or alert boxes. Regarding analysis, we worked through most of the bugs in the candidate creation and sending. However, in a handful of instances, character-escaping the response body behaves incorrectly leading to bad candidates.

A.2. Request Comparison

Comparing request parts, like the body, as string can be suboptimal as many differences in the bodies can completely shift values and render the diffing useless. So, before comparing requests to extract varying elements (our variables), we transform each part of the request (URL path, URL query, headers, and body) into a dictionary. Some parts like URL queries and headers are straightforward to represent as dictionaries. For URL paths, we divide the segments of the path and key them with `url_i`. The body presents the biggest challenge as it can be encoded into various formats or compressed. We cover the most widely encountered formats like `JSON`, `URL-FORMDATA`, URL-encoded `JSON`, URL-encoded `URL-FORMDATA`, base64 `JSON`, by detecting the format and transforming them into dictionaries. We later use `deepdiff` python library to get the difference between the two bodies.

A.3. Variable Filtering Heuristic

After extracting all variables in the requests in the form of dictionaries (`variable_name: variable_value`), we need to filter whether these variables are interesting (i.e., resource identifiers) or uninteresting (i.e., credentials or other random identifiers). We rely on heuristics and pattern matching to classify variables:

For URL paths, the names of the variables are `url_<i>` where `i` is the index of the segment in the path. For the rest of the components (headers, query, and body), we use the keys provided by the request (e.g., header name, query parameter name, body key if JSON, etc.).

We maintain three pattern lists: a *name ignore list (NIL)*, *name allow list (NAL)*, and *value allow list (VAL)*. We also create an *identity keywords list (IKL)* which contains the username, email, and other info of our users – the idea being that any variable including these, should be interesting to swap. All lists can be found in the code repository of the framework.

A variable is considered interesting to swap if:

1) the value matches the *IKL*.
2) the name DOES NOT match the *NIL*
    a) AND (the name MATCHES the *NAL* OR the value matches the *VAL*)
    b) AND the value is not a valid timestamp
    c) AND the value is not a floating point number

The following are example REGEX patterns we use to match the *NIL*, *NAL*, and *VAL*:

- *NIL*: `req(uest){0,1}[\-\._]*(S|s)ig` (matches request tokens and signatures)
- *NAL*: `list(Id|id|_id)` (matches resource list identifiers)
- *VAL*: `[0-9]{6,}` (matches numbers with 6 or more digits)

The full lists are available as text files in the project repository TODO, and can be easily updated to fit the needs of future studies.

Appendix B. Visitation Protocol

We communicated the following guidelines with researchers conducting the website visitation:

1) The goal of the interactions is to cover user-related actions with the website, this includes:
    a) navigating in user account and user profile editing
    b) likes, follows, bookmarking
    c) view followers, subscriptions, friends, ...
    d) uploading data
2) In case the action you will perform asks for additional credentials (verifying the password again), do not proceed and return to the previous state.
3) In case your action involves inviting a user or adding a friend, use the third account you are provided that is not used in any of the two sessions you have opened up.
4) **Do not perform interactions that interact directly with external users,** e.g., messaging an apartment owner, sending a friend request, submitting a form that requires other users to process and follow up on (e.g., schedule meetings) as they can be considered as spam.

5) Perform interactions at a slow enough pace that the follower browser can catch up, and the live-swapping workers can catch up.
6) Avoid clicking on the following:
    a) window popups, e.g., `alert()`
    b) `iframe` or embedded third-party content
7) **Always remain on the first browser tab for each browser.**
    a) In case the website opens additional tabs when navigating, then
        i) copy the URL from the second tab back to the first tab in the leader
        ii) close the second tab for both the follower and the leader
8) Keep performing interactions until you think you covered all possible interaction types possible in the site for a user, ex: if liking an item is possible, you only need to do it once.
9) If your interaction changes the account states for both users, make sure to revert this action, e.g., you check "make my profile public" and save, then you need to un-check it and save again.
10) If you want to change personal information for both users, which has to be different for each user do the following:
    a) change the value in the input field of the leader. Your value will be copied to the follower which we don't want.
    b) change the value in the follower browser. Now the leader and follower have different values.
    c) click "save" on the leader.
    d) make sure to revert the action if it changes the state.

Appendix C. Additional Vulnerability Examples

***Vulnerability 4: RedactedPublisher.com* .** The last service we demonstrate is an online publisher that people can use to add articles to their websites. When a user creates a new document draft, it is accessible for the user via a newly generated 11-character long ID: */worflow/drafts/<ID>/info*. However, unlike other similar URLs on this website, the server does not check the authorization when accessing this URL and returns the draft. Other URLs instead return a message containing the current user's and the author's ID.

A *targeted attacker* knowing this ID could thus retrieve the current status of a document draft on this platform.

Instead of basing the authorization check on the ID, we recommended that the vendor enforce authorization checks based on the user's current session, as it is done by other endpoints. Additionally, to prevent the leak of the author's user IDs, we also suggested using a generic error message rather than a detailed one.

Similar to the previous examples, this finding benefited from our framework with which we could create new articles without harming other users.

Appendix D. Deployment, Scalability & Usability

In this section, we share details about the usability of the VSF and how it can be deployed.

We implemented the VSF to be as user-friendly as possible and we found that the involved student helpers did not find any issues learning how to use it. On average, participants spent 7.75 minutes (σ = 3.9) on each site and performed 54 interactions on average, naturally varying greatly between sites (σ = 29). Additionally, the GUI analysis app accelerates candidate analysis and uncovering vulnerabilities.

We implement the VSF over multiple inter-connected Docker containers (e.g., a manual visitation container, swapping container, automated visitation coordinator, etc.). This helps with deploying the framework anywhere without worrying about dependencies and special requirements. This also means that the framework scales easily with the increased participants, by simply booting up new manual visitation containers. We also use VNC, to reduce the requirements of participants, only accessible to involved researchers through a central IP in the university's network.

Appendix E. Recommendations For Server-Side Developers

In this section, we showcase general primitives from the endpoints we analyzed that could prevent many of the *AC* violations. These guidelines are agnostic of the application or infrastructure developers use.

P1. Infer all user-specific identifiers from their credentials rather than passing them through in the request. For example, rather than using `/profile/<username>`, developers could have a generic `/profile` and extract the username from the credential, like `request.user` in Django. This prevents many opportunities, where an authenticated user can impersonate another.

P2. Separate shareable resource identifiers from internal identifiers. For example, for a website managing PDF documents, each document can have an internal identifier used for internal APIs and a public (*possibly temporary*) public identifier used for shareable links for this PDF. Many file-sharing services like Microsoft OneDrive implement such a technique [41].

P3. Use POST request instead of GET for requests with sensitive identifiers, and move them from the URL to the body. This is a simple technique that protects users from accidentally pasting the URL somewhere it should not be or screenshot leaks.