# SECURITY WEAKNESSES IN E-COMMERCE PLATFORMS

ROHAN PAGEY

A THESIS

IN

THE DEPARTMENT OF

CONCORDIA INSTITUTE FOR INFORMATION SYSTEMS ENGINEERING

PRESENTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF MASTER OF APPLIED SCIENCE

IN INFORMATION SYSTEMS SECURITY

CONCORDIA UNIVERSITY

MONTRÉAL, QUÉBEC, CANADA

JANUARY 2023

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis prepared

|  |  |
|---|---|
| By: | **Rohan Pagey** |
| Entitled: | **Security Weaknesses in E-commerce Platforms** |

and submitted in partial fulfillment of the requirements for the degree of

**Master of Applied Science**
**in Information Systems Security**

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

|  |  |  |
|---|---|---|
| Dr. Mohsen Ghafouri | ______________________________ | Chair |
| Dr. Mohammad Mannan | ______________________________ | Supervisor |
| Dr. Amr Youssef | ______________________________ | Supervisor |
| Dr. Mohsen Ghafouri | ______________________________ | Examiner |
| Dr. Carol Fung | ______________________________ | Examiner |

Approved by______________________________________________
Dr. Zachary Patterson, Graduate Program Director

January 23, 2023 ______________________________________________

Dr. Mourad Debbabi, Dean
Gina Cody School of Engineering and Computer Science

# ABSTRACT

Security Weaknesses in E-commerce Platforms

Rohan Pagey

Software as a Service (SaaS) e-commerce platforms for merchants allow individual business owners to set up their online stores without any coding, or procuring any software/hardware. Prior work has shown that the checkout flows of such e-commerce applications are vulnerable to different kinds of logic bugs such as parameter tampering or workflow bypass, with serious financial consequences, e.g., allowing "shopping for free". In this work, we first present a list of typical operations for such platforms, showing that there are several more functionalities beyond the check-out process, which can also lead to serious security consequences. We then leverage the fact that such platforms now heavily incorporate API requests and GraphQL calls (emerging) to design a semi-automated security analysis framework. We use this framework to analyze 32 representative e-commerce platforms (including 8 open-source ones) for seven different vulnerability categories; such platform host over 10 million stores as approximated through Google dorks. We uncover several previously unknown vulnerabilities with serious consequences, e.g., allowing an attacker to take over *all stores* under a platform, and listing illegal products at a victim's store—in addition to "shopping for free" bugs, without exploiting the checkout/payment process. We found 12 platforms vulnerable to store takeover and 6 platforms vulnerable to shopping for free, affecting thousands of stores (49000+ for store takeover, and 28000+ for shopping for free, as approximated via Google dorks). We have responsibly disclosed the vulnerabilities to all affected parties: two vendors have fixed the issues and four are still working. We have also requested four CVEs (amongst the 8 open source projects), and three CVEs have been assigned.

# Acknowledgments

I would like to thank my supervisors, Dr. Mohammad Mannan and Dr. Amr Youssef for their constant support and guidance throughout this project. Their continued support gave life to this project and made this research possible. I would also like to express my gratitude for their patience, motivation, enthusiasm, and immense knowledge. I am incredibly lucky to be able to work under the close guidance of my supervisors who inspired me with bright ideas, helpful comments, suggestions, and insights which have contributed to the improvement of this work. I am also thankful to them for the substantial financial support that eased the financial burden while doing this research.

I would also like to thank my peers at the Madiba Security Research Group for sharing their knowledge and experience and being there beside me on my rainy days. I learned a lot from everyone, especially, Sajjad Pourali, Bhaskar Tejaswi, Xiufen Yu, and Pranay Kapoor. I feel honored to have worked with them. Special thanks to all professors at CIISE for providing me with the opportunity to learn in a positive learning environment and made me more and more interested in all aspects of systems security.

Lastly, I would like to thank my family and friends. This journey would not have been possible without their encouragement and support.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

## 1.1 Overview

Not many online shop owners are well versed with technologies, such as developing and maintaining a website or a mobile app to sell their products. Hence, online shop owners often rely on Software as a Service (SaaS) e-commerce solutions that enable creating/-managing online stores *quickly* and *easily*. According to recent reports, Shopify, a Canadian SaaS e-commerce platform, is used by 1.75 million merchants, generating a revenue of around $2.93 billion [6, 38]. The popularity of such platforms has increased over the past few years, as it rejuvenated brick and mortar businesses, which were interrupted due to the COVID-19 pandemic [9, 16].

A SaaS e-commerce platform is different than a normal shopping website in terms of complexity, the number of components, and their end-users. For example, there are more roles in a SaaS e-commerce platform (than only customers in a shopping site); e.g., a store owner who creates a store, and the owner can then create more accounts under the same store and assign them a lower role (e.g., order manager, and seller [41]). Such added complexity often results into serious security vulnerabilities, as evident from recent reports [14, 35], exposing customer data or causing account takeover.

## 1.2 Motivation

Recent academic studies revealed several security and privacy issues related to business logic in e-commerce content management systems (CMS) [33, 43]. Notably, Wang et al. [45] performed security analysis of Cashier-as-a Service-based e-commerce applications, and found that leading merchant applications and online stores contain logic flaws that can be exploited to shop for free. Their follow-up work [33] explored more test cases (generated and executed automatically) in terms of logic vulnerabilities.

Existing work extensively focused on the checkout process or the payment modules from a customer's viewpoint; however, there are more complex and vulnerable components/features with serious consequences than just the payment module. GraphQL APIs (providing more powerful functionalities than traditional REST APIs) in e-commerce platforms have also not been analyzed as they are relatively new (stable release 2018).

## 1.3 Problem Statement

This work intends to answer the following three research questions. First, whether operations other than checkout can be exploited in e-commerce platforms to shop for free, and if so, what are the underlying vulnerable patterns? Second, are there security consequences beyond shopping for free affecting these platforms? Third, are the platforms' primary operations affected by unauthorized read/write access control issues and the use of GraphQL?

## 1.4 Summary of Our Approach

As a first step to answer our research questions, we identify a list of common operations for SaaS e-commerce platforms — by analyzing the network traffic, exploring the available functionalities, and examining the available documentation. We then identify six categories of vulnerabilities, and four categories of major security consequences. The vulnerabilities we consider are mostly based on past work in e-commerce platforms (e.g., [33, 43, 45]), web attacks (e.g., [42]), OWASP API top-10 [23], and GraphQL vulnerabilities [27].

We design a security evaluation framework to analyze the identified vulnerabilities in SaaS e-commerce platforms (both websites and Android apps), and open source e-commerce tools for setting up online stores. We collect network traces while interacting with the application, and then identify vulnerabilities by considering multiple types of API requests and GraphQL calls (as used to enable complex functionalities). We rely on automated tests to detect broken authentication and access control issues [24], and semi-automated tests to detect cross-site request forgery [25] and insecure GraphQL batching [27].

Our approach to detect broken authentication and access control is inspired from an open source tool *Auth analyzer* [34], which we customize for automatic identification of different API requests in an e-commerce platform. In particular, we identify some operations that generate a specific type of API requests which must be sent only once to the server for accurately identifying access control vulnerabilities. We create a list of common keywords for such operations to support automatic detection of such operations or API requests. Then, we automatically tamper with HTTP requests, based on a predefined pattern (i.e., session token replacement), and determine the presence of broken access control by evaluating the differences between tampered and original server responses. We also check for the presence of redundant parameters in a request as they can be used to bypass improper access control implementations.

We save the browsed and tampered traffic in a file that we refer as a flow file in our framework. To detect other classes of vulnerabilities, we apply our analysis on top of the already saved flows. We also consider three other vulnerabilities, namely injection attacks [28], mass assignment [15], and weak rate-limiting [29]; we carefully analyze them manually (due to ethical concerns) while browsing the platform. We note that our chosen categories of vulnerabilities are generic enough to be applied in other web services, and our automated approach for checking broken authentication and access control issues can be used to evaluate any web application with multiple types of user/role accounts.

We implement several components in our framework on top of MITM proxy [20]. For the automation of broken authentication and access control analysis, one of the challenges we encountered was removal of unnecessary URLs with no effect on the access control. To solve this problem, we create three filters to apply on top of the flagged alerts. First, a *least-role filter* removes the intended access of users having a lower role; the intended access is represented using a list of URLs and eliminated from the flagged alerts. Second, an *option-method filter* eliminates API calls with HTTP OPTIONS method, since such calls do not access or update any resource; and third, a *static-URL filter* removes the URLs pointing to static resources (e.g., JavaScript, PNG files). We apply these filters in conjunction, and automatically remove unrelated API endpoints by more than 50%, thus lowering the scope for manual analysis. At the end, all our alerts for access control vulnerabilities are true positives. Manual analysis is done only to assess the impact of the identified exploitable alerts. Note that we cannot assess our false negatives (i.e., the missed vulnerabilities) due to the absence of ground-truths. We apply our framework on 32 e-commerce platforms (24 SaaS-based and 8 open source projects), hosting approx. millions of online stores; see Tables 2 and 3, for the Google dorks we used and the platform URLs. From the analysis of the 32 SaaS platforms, our framework revealed various types of flaws with serious consequences. Examples include: returning valid authentication tokens even for incorrect passwords, and allowing the addition of unauthorized accounts to another merchant's store—both leading to platform-wide store takeover; allowing users to arbitrarily change the balance in their accounts or issuing refunds, checking for *any* authentication token, i.e., not enforcing the token assigned to a specific account after login—leading to shopping for free bugs.

## 1.5 Contributions

1. We develop an experimental framework for systematically evaluating security vulnerabilities in e-commerce platforms, both in storefronts (visible to regular users)

and store dashboards (visible to merchants). Our framework comprehensively assesses all common functionalities (beyond checkout/payment) offered by leading e-commerce solutions—closed-source SaaS products, as available through web services and Android apps, and open source solutions. Our tests include traditional web vulnerabilities as applicable to these e-commerce operations, as well as tests for the newly-adopted GraphQL APIs.

2. We apply our framework on 24 representative SaaS based e-commerce platforms and 8 open-source solutions for merchants, and reveal a total of 38 serious security vulnerabilities, some of which allow *platform-wide* (i.e., affecting all stores) full store account takeovers and shopping for free. Our results highlight that most platforms fail to adequately preserve the security of their users – both for store owners and customers.

3. 12/32 e-commerce platforms have multiple store administrative functionalities vulnerable to access control flaws, improper input validation, and cross-site request forgery—allowing an attacker to takeover *all* the stores under them—a total of 41,210 stores, as approximated via Google dorks. An attacker can also access customer details of all users: email, physical address, mobile phone number, and past orders.

4. 6/32 platforms (*Swell, Storehippo, Okshop, Shoppiko, Branchbob, Bikry*) do not protect several e-commerce operations (e.g., profile/storefront management, order and coupon handing), allowing an attacker to shop for free in all stores under them (28,817 stores, as approximated via Google dorks).

5. The open-source platform *nopCommerce* (v4.50.2) is vulnerable to improper access control, allowing an attacker to modify every customer's address in a store. The vendor rolled out a new version, fixing the vulnerability in two days after our disclosure. We are assigned a CVE for this.

6. A vulnerable GraphQL query in *WooGraphql*, which is an extension for *Woocommerce* (estimated to be used by 5 million stores as of 2022 [3]), allows an attacker to collect all existing coupon codes of a store and use them to shop for a lesser price. We are assigned a CVE ID for this.

7. Another open-source platform, *AbanteCart* (v1.3.2) is vulnerable to reflected cross-site scripting and SQL injection, enabling an attacker to takeover a victim's session by crafting a malicious URL and luring the victim to click on it (in XSS), and dumping the back-end database (by exploiting SQLi). We are assigned two CVEs for these.

## 1.6 Thesis Organization

The rest of the thesis is organized as follows. In Chapter 2, we first present an overview of common e-commerce entities and operations. We describe our threat model, scopes of considered attacks and types of attackers. We also discuss the prior research work relevant to our study. In Chapter 3, we introduce our framework and present the methodology we use to perform a security evaluation of e-commerce platforms. In Chapter 4, we discuss about the platforms we selected for our research as well as the implementation of our framework.

In Chapter 5, we present the experimental results of our analysis, discuss the impact of our findings, and evaluate the performance of our framework in terms of efficiency, and false positives. In Chapter 6, we discuss the limitations of our work and include key insights from our work and recommendations for platform developers and users. Finally, we present our concluding remarks and future work.

## 1.7 List of Publications

The following publications [31], [17] resulted from the research work performed during my master's program. Most of the work presented in this thesis has been peer-reviewed and accepted in the first article.

- Rohan Pagey, Mohammad Mannan, and Amr Youssef. All Your Shops Are Belong to Us: Security Weaknesses in E-commerce Platforms. In proceedings of the ACM Web Conference (TheWebConf'23), April 30- May 4, 2023, Austin, Texas, United States.

- Pranay Kapoor, Rohan Pagey, Mohammad Mannan, and Amr Youssef. Silver surfers on the tech wave: Privacy analysis of Android apps for the elderly. In 18th EAI International Conference on Security and Privacy in Communication Networks (Securecomm 2022), Kansas City, United States, October 2022.

# Chapter 2

# Background

In this section, we summarize key SaaS e-commerce components, their typical operations, and define our threat model.

## 2.1 SaaS E-commerce Entities and Operations

1. **Merchant** - A merchant is the central role in SaaS e-commerce platforms. They can build and customize an online store and are responsible for managing users, tracking payment status, recording order details, finalizing orders, and shipping products (or providing services) to customers. A merchant account has the highest role in a store (i.e., a store admin), which if compromised, will lead to a full store takeover.

2. **Merchant user** - A merchant user is defined as an entity having a valid role in a store (e.g., store manager, reseller, order manager), added by a store's merchant to support various store operations. A merchant user has a higher privilege than the customers.

Example roles of some merchant users include: store manager, reseller, and order manager.

3. **Customer** - A customer is a self-registered entity in the merchant's store. Depending on the SaaS platform, a customer can either register to a specific store, or create an account at the platform, which can be used at any other store. A customer can browse products, initiate the checkout process and make a payment to the merchant. A customer account does not have any role assigned to it by a store's merchant.

4. **CMS server (CS)** - A CMS server hosts the store content. It is owned by the SaaS platform and the price for hosting is usually included in the subscription fees (paid by the merchants).

Key components in a SaaS e-commerce solution include: the *CMS server (CS)*, the *store dashboard* (admin panel for managing a store by the merchant, and merchant users), and the *storefront* (customer login and order placement). We identify the following main functionalities from several representative platforms.

1. **Store creation** - As illustrated in Fig. 1, When a merchant wants to start an online e-commerce business, she selects a SaaS provider and requests for a personalized store. The request usually contains parameters such as store name, merchant email, and password. The CS generates a random token (e.g., OTP, custom link), and sends it via SMS or e-mail to verify the request; if successful, the store front/dashboard URL and a store ID is created and returned to the merchant.

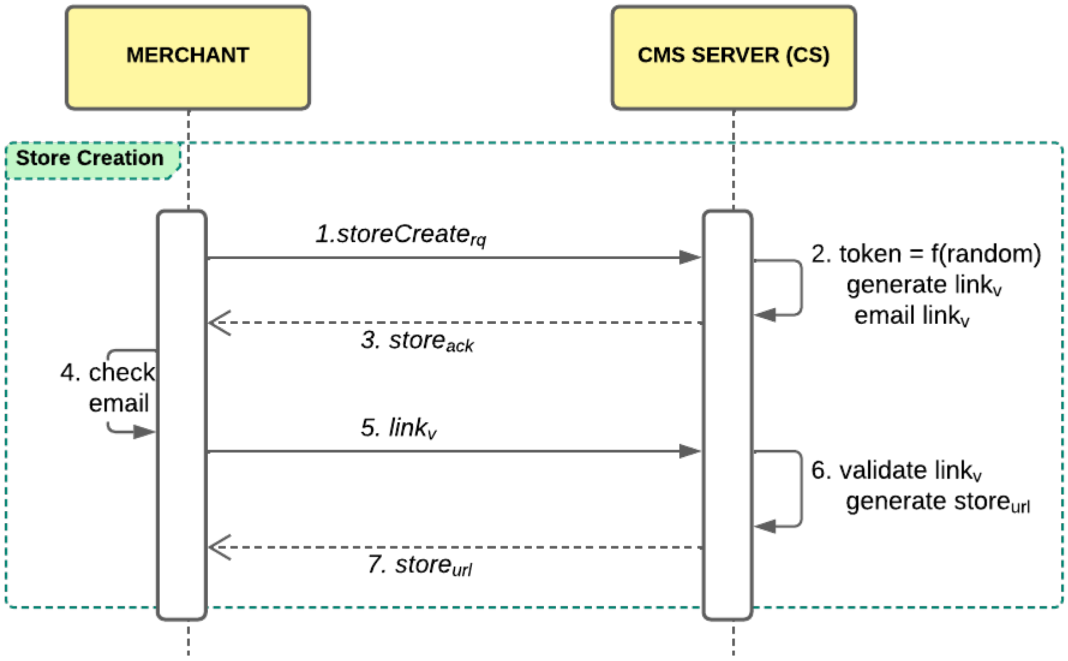

Figure 1: Merchant Store Creation Process; $StoreCreate_{rq}$: Store registration request; $link_v$: OTP/verification link; $store_{url}$: Store dashboard URL

2. **Merchant and customer authentication** For both user types, the authentication steps are mostly the same, requiring user IDs and passwords (and store IDs for merchants), which are verified by the CS; if successful, an authentication token (e.g., a session cookie, JSON web token) is sent to the merchant/customer. A merchant's authentication request originates from the store dashboard, as opposed to the storefront, for a customer, and the authentication logic may differ between these cases. For subsequent requests, both merchants and customers must include their authentication tokens.

3. **Profile management** - For a store merchant, this operation represents modifying any

of the store settings, such as adding new roles to the store. For a customer, it represents the functionality to change her profile parameters, such as physical address, email.

4. **Storefront management** - This interface allows merchants to set up their stores (e.g., add/remove products) using an interactive drag and drop menu bar, and adding HTML code, images, or extra pages.

5. **Order processing** - This operation contains four sub-operations—order create, read, update and delete. From a storefront URL, a customer can create an order request (with payment information, unique item IDs) and forward it to the server along with her authentication token. Upon order verification and successful payment, the merchant generates an order approval request message, to be verified by the CMS server. The product is shipped after the verification is successful (may be manually reviewed by merchants).

6. **Coupon handling** - This also involves 4 sub-operations—coupon apply, modify, add, and delete. The apply coupon operation is intended for customers, and the others are meant for the merchant.

As apparent from the overall functionalities, the CMS server is responsible for handling the authorization of every request it receives, and it also controls the information returned in the response. Improper implementations of these two items often result in critical access control vulnerabilities.

## 2.2 Threat Model and Attackers' Goals

We consider the following two types of attackers: (i) *platform-wide attacker*, who can attack *any* store on a vulnerable platform, simply by knowing the public victim-store URL; and (ii) *store-specific attacker*, who can only attack a store by knowing some non-public, (high-entropy) store-specific parameters, e.g., by being a low-privileged merchant-user in a particular victim store. Compared to the platform-wide attacker, a store-specific attacker is very limited in scope (i.e., attacking only a specific vulnerable store with the knowledge of some store-specific parameters). We consider attacks that can be launched remotely, with or without requiring any user-interaction; we also require no physical access (merchant device or SaaS platform infrastructures). We assume that attackers can create their test merchant and merchant user accounts in a target SaaS platform without paying any significant fees, ideally no fees. (Note that such payments may not deter a motivated attacker, especially if the returns are much higher than the imposed fees.) Also, some attacks can be carried out using a regular customer account. We do not consider network attackers as most of network issues can be solved by implementing HTTPS properly.

We consider four major attack goals that have significant security consequences: (1) *full store account takeover* to perform all or most actions that only a store merchant is authorized to do; (2) *store defacement* to control the contents on a storefront without full store takeover (e.g., add/remove products); (3) *shopping for free* by modifying existing product parameters, including a product's price, or any applicable discount/coupons; and (4) *sensitive information disclosure* to compromise user/merchant privacy (e.g., historical orders) and to exploit the exposed information for other attacks (e.g., shop for free). We also consider two minor attack goals: (5) *denial of service* to exhaust the server's resources by exploiting a vulnerability in the website; and (6) *free membership* to upgrade to a paid plan offered by the platform, without paying any fee.

## 2.3 Ethical Consideration and Responsible Disclosure

The testing steps are performed only on our own accounts. We refrain from accessing sensitive information of other users. We also contacted the security team and the developers of the analyzed platforms, and shared our findings with proof-of-concept attacks, security consequences, and guidelines on possible fixes. We have been gradually communicating (several times for some) with the companies in the past 3 months as the vulnerabilities are found. *Sellfy* and *nopCommerce* have fixed our reported issues within two days of reporting. Four companies are still investigating the issues, namely *Swell*, *McaStore*, *Lovelocal* and *WooGraphql*. Note that we do not report vulnerabilities which are listed as "known issues" in the platform's vulnerability disclosure page.

## 2.4 Related Work

Prior studies [33, 43, 45, 47] mainly focus on the security of the check-out processes in an e-commerce storefront (visible to customers), and reveal numerous logic flaws when integrating services of third-party cashiers. Wang et al. [45] conducted the very first detailed study on Cashier-as-a-Service based web stores, and found serious logic flaws allowing a malicious shopper to purchase items for free. Xing et al. [47] propose a proxy model for testing stores that rely on third-party checkout integration and single sign-on authentication. Sun et al. [43] propose a static analysis mechanism for detecting logic vulnerabilities in e-commerce applications while focusing on logic flows of the check-out process. In contrast to these studies, we focus our security analysis on the merchant dashboard and other store management features, which revealed equally important or even more serious security flaws (e.g., shopping for free vs. platform-level attacks affecting all stores).

Sun et al. [44] analyze scam operations on merchant sites, including the refund process scams, demonstrating an attack vector (involving social engineering) to shop for free by scamming the merchant. We also found similar issues in *Storehippo*, where an attacker can directly make an API call to exploit the refund API without requiring any victim interaction or social engineering.

Giancarlo and Davide [33] propose a black-box approach to detect logic vulnerabilities in web applications, using an application-independent model interference technique. They capture network traces between the client and server, and create a navigation graph from the traces. Then they extract access control patterns related to the underlying application logic by applying some heuristics. Then test cases and attack patterns are created to find parameter tampering and workflow bypass vulnerabilities (in open-source e-commerce solutions). However, as the authors mention, their approach is not designed to detect other types of logic vulnerabilities, e.g., unauthorized access to resources. As their test cases involve malformed operations, they evaluate only the open-source platforms. Similar to past work, they also applied their framework on the check-out process in a customer-facing storefront.

Security concerns pertaining to the payment systems [19, 22, 48] in e-commerce have also been studied extensively in past research. Yang et al. [48] analyze major Chinese third-party in-app payment systems, which are integrated in many Chinese Android apps. They report serious implementation flaws in the implementation code of hundreds of apps, and also reveal design issues in the payment SDKs. These flaws generally allow an attacker to shop for free in various ways. Beyond large commercial payment services, Lou et al. [19] conducted a systematic study on 35 Chinese personal payment systems, which are also integrated with many apps and web services. They found at least one security vulnerability in each of the analyzed payment systems, resulting into financial losses for payment systems, businesses relying on them, and users.

Researchers have also developed generic open-source automation tools [21, 34, 37] for detecting unauthorized read/write vulnerabilities in web applications. Generally, these tools replace session tokens or some parameters in the original request, and then detect unauthorized access by comparing the differences between the modified and original responses. However, some tools [21, 37] do not handle the requests that generate error upon multiple calls. Auth-analyzer [34] is an improvement over the other tools as it supports all types of requests, although manual effort is needed for handling the different requests. Our approach for detecting broken authentication and access control issues is based on auth-analyzer, and we automate distinguishing different types of requests based on e-commerce related keywords (see Table 1).

In summary, we conduct a comprehensive, systematic study of security vulnerabilities in e-commerce platforms, covering both the storefront (exposed to regular customers) and store dashboard (exposed to store operators only). We show that there are more vulnerable components than just the check-out process, even within a storefront (such as modifying customer details, adding coupon code). Our framework supports websites, Android apps, and open-source products, as popular e-commerce platforms make use of all these solutions. Our analysis therefore sheds light on the broader picture of security vulnerabilities of e-commerce platforms. Since the logic vulnerabilities in check-out process and the payment systems have been extensively covered in prior work, we exclude them in our work. Rather, we show new attack vectors that lead to the same consequence as a vulnerable check-out process such as shopping for free. Our attack vectors would work even if the check-out process is securely implemented.

# Chapter 3

# Security Analysis Framework

We perform automated, semi-automated and manual analysis on top of the HTTP traffic to identify security vulnerabilities in the e-commerce platforms used by merchants and customers. The security issues tested are inspired by previous research in the e-commerce security [33, 42, 43, 45] and refined by us to cover more test cases and vulnerabilities. For this purpose, we systematically review academic literature in web security [4, 7, 12, 18], and multiple non-academic resources [1, 23, 30]. We consider the vulnerabilities that can be exploited by a remote attacker as per our threat model, which are relevant to SaaS e-commerce.

**Overview.** For each platform, we first create a merchant account and capture the HTTP/S traffic as we explore the store dashboard web application (or an Android app, if available). We let the traffic pass through MITM proxy and manually analyze the live traffic for injection attacks, mass assignment, and weak rate limiting. Second, we run a session modification component in the background (with the authentication token of an attacker merchant account, which we also create), while the browsing merchant explores the platform interface. The session modification component generates a flow file, storing all the HTTP requests and responses generated by the modified and original sessions. Thereafter, we apply a set of active and passive analysis techniques on top of the flow file to evaluate different security issues; see Fig. 2. For testing the open source software for merchants, we download their source code and follow their documentation to deploy them locally on our lab systems.

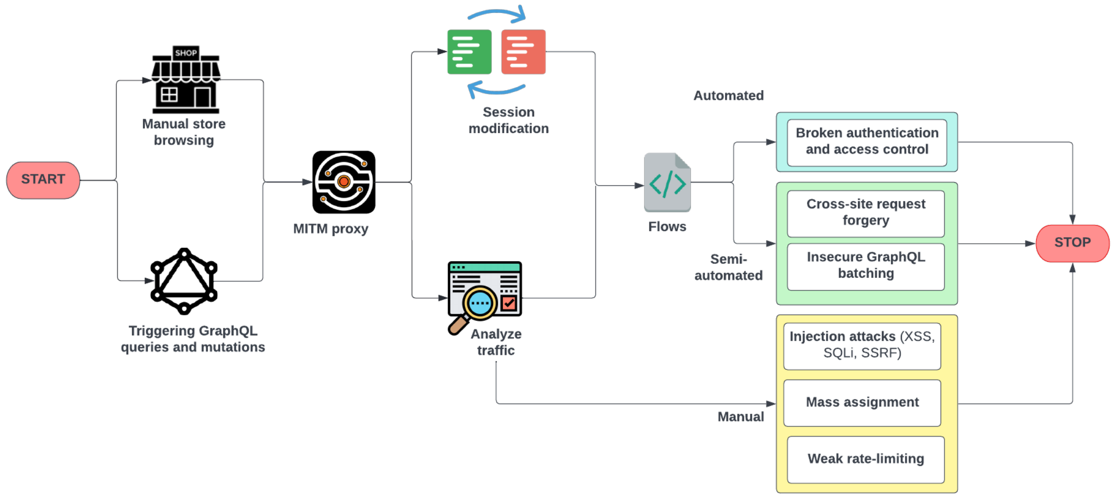
START
Manual store browsing
Triggering GraphQL queries and mutations
MITM proxy
Session modification
Analyze traffic
Flows
Automated
Semi-automated
Manual
Broken authentication and access control
Cross-site request forgery
Insecure GraphQL batching
Injection attacks (XSS, SQLi, SSRF)
Mass assignment
Weak rate-limiting
STOP

Figure 2: Overview of our proposed security analysis framework

## 3.1 Broken Authentication and Access Control

We first create two merchant accounts (i.e., two different stores): a browsing merchant and an attacker merchant. We then login as the attacker merchant to receive her authentication token. We supply this token to our session modification component, which runs in the background while we explore a store UI as a browsing merchant. We also save the browsing merchant's session in a flow file for further analysis.

We divide all API requests into two types. (1) Requests that can be successfully called multiple times without any modifications in the request body while producing no errors; e.g., a GET request to view user details or a POST call to update a customer name. For these APIs, we first send the original request to the CMS server to get the expected response, and then forward the modified request (by swapping merchant/attacker authentication tokens). Receiving the same response for both requests indicates an access control vulnerability. (2) Requests that generate errors when called more than once. For example, a DELETE API call to remove an existing customer can only be used successfully once because the server would reply with e.g., "customer already deleted" message upon any further calls. Similarly, a POST request to register a user can only be called once (further calls would generate "user email already exists" from the server). Requests to create an object also fall under this type, as they need unique parameters on every call, e.g., each coupon object would need a unique coupon code. Responses for the original and modified requests in this case might not be the same (even if there is an access control vulnerability). Hence we only send the modified request, and determine an access control violation by checking the status code (e.g., `200 OK`) and the response's content-length. We automatically distinguish these requests using specific keywords and method types (detailed in Sec. 4).

Besides testing broken access control issues with the attacker merchant role, we also use other user roles. We supply a target role's (e.g., order-manager) authentication token instead that of the attacker merchant's, and automatically replay the original requests from the saved flow file (store merchant). This creates HTTP/S traffic for the user with a lower role, which is saved in a new flow file. We then compare the newly created flow file against the original one to identify any access control issues. We also replay the saved original requests (from the merchant flow file) by stripping off the authentication headers, and compare the replayed and original traffic to detect broken authentication issues.

We create three types of filters and apply them on the flagged alerts to remove the non-exploitable API endpoints.

1. *Least-role filter*: Since a user with a lower-privileged role would have some functionalities common to that of a store merchant, we need to remove the API URLs that the lower-privileged user is intended to access. In order to know the intended access, we browse as the user having the least possible privileges (e.g., the merchant user role with the least features, if the platform provide adding such merchant users; otherwise, simply the customer role) to generate another flow file; the API endpoints present inside the least-privileged user's flow file are then eliminated from the flagged alerts. Using the least-privileged role helps to cover vulnerabilities in more API endpoints since the difference in functionalities between the store merchant and least-privileged role is maximum (as compared to other roles). Consider an example where a store merchant and a merchant-user (not the least-privileged role) can access 5 and 3 API endpoints, respectively. Using this merchant-user role as a filter would help to uncover vulnerabilities in 2 (5-3) API endpoints, since 3 endpoints are intended for the merchant-user here. Comparing this to using the least-privileged user (e.g., shipping manager), which has access to only one API endpoint, this filter would now uncover vulnerabilities into 4 endpoints, and hence ensure maximum coverage for vulnerable

APIs.

2. *Option-method filter*: We remove the API requests with the HTTP “OPTIONS” method,[1] as this method is not used for any read/write operations. It is common for these methods to have same response code—irrespective of the request token provided (attacker/victim).

3. *Static-URL filter*: We filter out the URLs which contain static resources such as images (e.g., jpg, png), CSS, PDF, Javascript, Xml and Svg media files. While this filter can introduce some false negatives in case of private files on a website, but we note that this is not the case with a SaaS e-commerce platform in general, as they do not usually have the functionality of having private static files, e.g., a private messenger app where users can share private files or an app where users can upload their PDF documents.

Creating and saving the original browsing traffic into a flow file helps us to run different automation scripts on it (e.g., replacing a request parameter and replaying a modified request), without having to browse again. It also allows us to check the responses (from all types of requests) for sensitive parameters, e.g., email, password, OTP, API key, and access token. Such exposed parameters from request endpoints with vulnerable access control, can result into account takeover vulnerabilities. We also analyze the network traffic inside flow file further to detect other vulnerabilities and missing security best practices, as discussed below.

[1] https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/OPTIONS

## 3.2 Insecure GraphQL Batching

GraphQL is designed to be an improvement over REST API by allowing the client to request only the needed data and hence solving the problem of over-fetching unnecessary data. However, there are some inherent features offered by GraphQL, which might make it more vulnerable than a REST API, e.g., batching queries (read operation) and mutations (write operation). Batching allows GraphQL to group multiple requests into one and make a single request containing all the combined data. Secondly, GraphQL also enables nested queries with a circular relationship, i.e., queries that reference each other. These two features can be combined to cause denial of service attacks (using nested/circular queries) and brute-force attacks, specially in case of authenticating mutations.[2] We note that a typical rate-limiting mechanism for regular APIs, would not stop such attacks as those mechanisms are designed to block execution of thousands of requests in a short-time frame but they would not detect a single request containing thousands of operations. Lastly, the errors returned from a failed GraphQL query are very descriptive by default. If there are no custom errors defined in the application, triggering a wrong GraphQL query might reveal correct parameters required to make a successful call.

First, we detect if the platform under testing is using GraphQL. We do this by automatically searching the flow file for the presence of standard GraphQL endpoints.[3] We also search on Google for any GraphQL documentation for the e-commerce platform under testing. Then, we then use an open-source GraphQL auditor tool [8] for detecting batching

[2]A mutation that is responsible for authenticating a user.

[3]Endpoints = ['/graphiql', '/playground', '/console', '/graphql']

attacks and circular queries. We note that the script only uses introspection queries [13] (which are generally not resource intensive). However, we further exclude some payloads (e.g., circular fragmentation queries) that can still be problematic for a server. We also manually probe the GraphQL queries to trigger descriptive errors and further analyze them to check the impact.

## 3.3 Injection Attacks

In injection attacks, a malformed user input is sent as a request parameter to the server, which if processed, can cause an application to behave in an unexpected way. Depending on the type of malformed input, an attacker can run arbitrary JavaScript code in a victim's session (XSS), inject SQL queries to dump the application server's database (SQL injection), or use the e-commerce platform's server to port scan their internal network (Server-Side Request Forgery). First, for XSS, we inject a simple payload[4] in only those request parameters that are reflected in the response to determine the unfiltered characters. In case our payload appears as it is in the response, we use JavaScript `alert()` to confirm the vulnerability. We also test for stored XSS primarily in the functionalities which are modifiable by a customer, such as their profile form or adding a product review. Note that customer information including their profile attributes, placed orders, reviews etc, is accessible to a store dashboard. Thus if a customer-injected payload is executed in a merchant's session can allow the customer to access merchant dashboard and takeover the store. Second, for SQLi, we avoid testing it on SaaS platforms due to possible modifications on the

[4] `<>'"{}()test`

server's database. We only test these payloads [32] in open source e-commerce platforms in our local setup. Third, for SSRF, we observe that there are very limited requests in an e-commerce platform accepting URLs as an input parameter. In particular, this test case is only applicable in the *webhook*[5] functionality offered to a store merchant. For detecting SSRF, we set up a public server using our lab machines and inject the link and different ports into the webhook URL. Upon triggering the corresponding webhook action, we check our server's log for any request originating from the e-commerce server's IP.

Due to ethical concerns, we refrain from using any automated scanning tools to detect injection attacks in SaaS platforms. For the open source platforms, we use XSStrike [39] and SQLMap [40] to automate testing for XSS and SQLi attacks, respectively.

## 3.4 Mass Assignment

Mass assignment vulnerabilities occur when a server-side object's sensitive properties are inadequately protected; e.g., a developer defines a list of object properties that a user is not privileged to modify, but omits some sensitive properties (e.g., `User.is_admin`). Note that an object is any entity with a valid unique ID (numeric or UUID) defined in the CMS system, e.g., products, users, and orders; each object may have multiple properties (read-/write permissions depend on the user roles). We assess mass assignment vulnerabilities manually (due to ethical reasons).

For every object, we first find a list of available properties by probing their API, checking the client-side code, or simply reading the API documentation. We mainly focus on

[5]Webhooks work as user-defined HTTP callbacks; see, e.g., https://zapier.com/blog/what-are-webhooks/

sensitive properties such as a product's price, customer's account balance, a merchant's subscription plan etc. After learning the available sensitive properties, we make an API request to modify the object, by including the known properties in the JSON request body. Before triggering the request, we change the property value according to its data type, e.g., considering an object property `user.balance`, we need to modify its value to be of a numeric data type. This is important because supplying a random value would trigger an error and would not update the property. After the modification request, we make a *GET* request to view the updated object property.

## 3.5 Cross-Site Request Forgery (CSRF)

A successful CSRF attack allows an adversary to trigger state-changing[6] requests in the victim's session.[7] In case of a merchant account, a site-wide CSRF would allow an attacker to takeover all stores on the site. Note that an attacker would first need the victim to click on her exploit URL to trigger a CSRF. We leverage the fact that a successful CSRF exploit requires three conditions to be satisfied (as per OWASP [26]): the client and server must not work with JSON data; there must not be any custom headers required in the request; and the request does not contain any anti-CSRF token. For each state changing request, we first check the content-type, and then search for anti-CSRF tokens in the whole request, based on the token name. As these token names are usually generic across different platforms, we can compare them exactly (literal matching), or using regular expressions for partial match.

[6]A request that can update some data on the server side–e.g., HTTP requests with POST, PUT, PATCH and DELETE methods.

[7]https://owasp.org/www-community/attacks/csrf

The list of regular expressions is initially taken from *CSRF Scanner*,[8] and then customized by us to add more token names.

## 3.6 Weak Rate-limiting

We perform limited password and OTP brute-force attacks (50 attempts) on our own merchant accounts; such attacks can facilitate account compromise. We also test for rate-limiting on API endpoints, the lack of which can cause DoS at the server-end. To keep the load on the server minimal, we test for the presence of defensive mechanisms by 1000 attempts in a quick succession within our account from a single computer. An attacker can perform application layer denial of service attacks locking out the victim accounts, and she can also brute-force weak passwords and OTPs leading to account takeover.

[8] https://portswigger.net/bappstore/60f172f27a9b49a1b538ed414f9f27c3

# Chapter 4

# Implementation and Platform Selection

**Implementation.** We provide the details of our framework's implementation in this section. We first bypass the SSL pinning (if present) for Android apps, by emulating the app on Genymotion [11] and then using the SSL-unpinning module of the Xposed framework [2]. We browse all the functionalities inside the Android app as a store merchant and capture HTTP/S traffic by setting up the mitmproxy [20], as we do for a website. For the open source e-commerce platforms, we download their source code and run it on our local server by following their instructions. Note that we follow a black-box approach for both commercial/open-source platforms. We run the session modification component in the background, while manually browsing the platform as the merchant. Since merchant is the highest role, browsing with it helps to cover more API URLs and generate a proper baseline request/response in the flow file. HTTP requests are extracted from these flow files and are used to filter some flows as discussed in Sec. 3.1. To avoid unforeseen consequences on live services, the following tests are done manually: injection attacks, mass assignment, and weak rate-limiting. CSRF, and insecure GraphQL batching are semi-automated, as they require removing false positives (albeit limited-number). Broken authentication and access control tests (our main focus) are automated (including false positive elimination); for impact analysis, we check manually.

**Session modification and flows generation.** We implement this component using Python as an add-on on top of mitmproxy [20]. It requires active session cookies as input from a merchant/merchant user/customer role (the same store or a different store). We use mitmproxy with the *-s* flag, and provide the path to our modification component. This flag allows additional add-ons as input, to customize the default behavior of mitmproxy.

The session cookies are collected by manually logging into a store and by analyzing the *Set-Cookie* response header from the Chrome browser's network tab (via the inspect element functionality). This component runs in the background while we browse the store, and tampers with the incoming requests from mitmproxy. In particular, we modify the session token of incoming requests with the one given as input. The session token can be in the form of a cookie, API key, or a different request header, and we determine the type of session implementation while manually browsing the platform. Specifically, we replay the authenticated flow repeatedly while removing a cookie value or an authorization header each time. For every replayed authenticated request, we check the server responses for `401` or `403` status codes, indicating that the cookie or header value removed is used as a session token.

To distinguish the request types (as discussed in Sec. 3.1), we check the HTTP method and the API endpoint of an incoming request. To identify requests that generate an error upon multiple calls, we check if the HTTP method is DELETE, or a POST/PUT request containing a specific keyword from our list (see Table. 1); if so, we forward only the modified request. Otherwise, we send both of the requests. The output from this component is a flow file generated automatically by the MITM proxy, containing both the original and modified HTTP requests.

| **Request description** | **Keywords** |
|---|---|
| Add a coupon code | coupon, promo, promotion, voucher |
| Apply discounts on cart | discount |
| Add pages to the store | page, blog |
| Add low-privilege users | customer, user, seller, staff |
| Create product inventory | catalogue, categories |
| Approve/deny an order | order, approve, refunds, currencies, returns, invoice |
| Add item into a wishlist | wishlist |
| Edit store URL | slug |

Table 1: Different requests for a SaaS e-commerce platform with the POST/PUT methods, which can produce errors in multiple calls, along with keywords to detect them.

**Broken authentication and access control.** For automated detection of these issues, we built an add-on on top of MITM proxy, programmed using Python and the Jinja templating engine [36]. Python is used to implement the core logic of access control detection, as described in Sec. 3.1. This component runs offline and does not generate any network traffic. Jinja templating is used to display the contents of the flow file in an HTML tabular format while highlighting the vulnerable endpoints. We mainly use HTTP status code and JSON response comparison to evaluate if the given API endpoint is vulnerable to access control.

**Platform selection and local setup.** For our evaluation, we chose 32 representative SaaS and 8 open source e-commerce platforms, including their websites and Android apps for the merchants, and customers. For the SaaS based platforms, we used “create e-commerce store” and “best e-commerce builder” as search terms on Google. Since *Shopify* is one of the largest platforms, we also used “sites like shopify” as the search criteria. We also relied on Wikipedia [46] for SaaS and open-source platforms (e.g., whether actively maintained and features offered). In the end, our selection of commercial web platforms includes a mixture of popular platforms (e.g., Shopify, BigCommerce), and some new/emerging platforms (e.g., Swell, Okshop). For Android apps, we used “sell online” and “digital shop” as search terms on Google Play Store and selected apps based on factors such as their number of installations, the number of stores, and availability of a trial plan (we also used the “similar apps” feature of the Play Store). For the open-source merchant platforms, we setup the test environment in our lab, by following the installation instructions available in their documentation.

We present the SaaS e-commerce platforms that we evaluated, along with their popularity and number of stores hosted, as estimated using Google dorks (excluding stores that use custom domain names); see Tables 2 and 3. The google dorks does not include stores which were set up by using a custom domain name and hence the number of stores only represent the lower bound.

| Platform | # of stores | Google dork | Website URL (app package) |
|---|---:|---|---|
| BigCartel | 3,750,000 | site:*.bigcartel.com | www.bigcartel.com (com.bigcartel.admin) |
| BigCommerce | 1,520,000 | site:*.mybigcommerce.com | www.bigcommerce.com (com.bigcommerce.mobile) |
| Bikry | 7,980 | site:*.bikry.com | bikry.com (my.bikry.app) |
| Branchbob | 9,300 | site:*.mybranchbob.com | www.branchbob.com |
| Crystallize | - | - | crystallize.com |
| Dukaan | - | - | mydukaan.io (com.dukaan.app) |
| Ecwid | 264,000 | site:*.company.site | www.ecwid.com (com.ecwid.android) |
| GraphCMS | - | - | graphcms.com |
| Gumroad | 335,000 | site:*.gumroad.com | gumroad.com (com.gumroad.app) |
| Lovelocal | - | - | www.lovelocal.in (chotelal.mpaani.com.android.chotelal) |
| McaStore | 402 | site:*.mcastore.co/ | mcastore.co (com.morecustomersapp) |
| Mozello | 19,700 | site:*.mozellosite.com | www.mozello.com |
| Okshop | 678 | site:*.okshop.in/ | www.okshop.in (in.okcredit.dukaan.onlineshop.nearme) |
| Saleor | - | - | saleor.io |
| Sellfy | 56,600 | site:*.sellfy.store | sellfy.com (com.sellfy.sellfyapp) |
| Shopify | 28,600,000 | site:*.myshopify.com | www.shopify.com (com.shopify.mobile) |
| Shopnix | - | - | shopnix.in (com.shopnix.shopnixadmin) |
| Shoppiko | 1,470 | site:*.store.shoppiko.com | shoppiko.com (com.shoppikoadmin) |
| Shopware | 5050 | site:*.shopware.store | shopware.com |
| Simvoly | - | - | simvoly.com |
| Storehippo | - | - | www.storehippo.com |
| Swell | 1680 | site:*.swell.store -inurl:status | swell.is |
| Volusion | - | - | www.volusion.com |
| Wix | 3,680,000 | site:*.wix.com | www.wix.com (com.wix.admin) |

Table 2: List of all SaaS platforms along with the number of stores hosted on them, as estimated using Google dorks on Oct 8, 2022 (no suitable dorks could be used for the ones represented with '-').

| Platform | Forks | Stars | URL |
|---|---:|---:|---|
| AbanteCart | 161 | 128 | https://github.com/abantecart/abantecart-src |
| Magento | 9039 | 10215 | https://github.com/magento/magento2 |
| Microweber | 750 | 2493 | https://github.com/microweber/microweber |
| nopCommerce | 4298 | 7579 | https://github.com/nopSolutions/nopCommerce |
| OpenCart | 4592 | 6592 | https://github.com/opencart/opencart |
| Prestashop | 4480 | 6760 | https://github.com/PrestaShop/PrestaShop |
| Saleor | 4732 | 16972 | https://github.com/saleor/saleor |
| Shopware | 774 | 2011 | https://github.com/shopware/platform |
| Sylius | 1978 | 7017 | https://github.com/Sylius/Sylius |
| WooGraphql | 102 | 528 | https://github.com/wp-graphql/wp-graphql-woocommerce |

Table 3: List of open source e-commerce platforms with the number of forks and stars indicating platform's popularity. *Saleor* and *Shopware* are both open source as well as SaaS based.

# Chapter 5

# Results

We detail our findings below; for an overview, see Table 5. For each vulnerability category, we provide a summary of the findings and their impacts. For several attacks, the victim store ID is required, which can be a store URL, a six-digit integer, or a UUID string. In all cases, it can be found by simply making a GET request to the public store URL. The notations used in Table 5 are as follows. ●: *platform-wide attacker* and ❍: *store-specific attacker* (⋆ implies victim interaction is needed); blank: not vulnerable. In instances exploitable by multiple attacker types, we consider the worst one (e.g., *platform-wide* attackers are worse than *store-specific* attackers), with the broadest scope.

## 5.1 Broken Authentication and Access Control

We found at-least one access control vulnerability in 13/32 platforms. We discuss here our findings by grouping them into the attacks and their corresponding vulnerable patterns; see

| Attacks | Vulnerable patterns | E-commerce operations | | | | |
|---|---|---|---|---|---|---|
| | | Authentication | Profile management | Storefront management | Order processing | Coupon handling |
| Store takeover | OTP/token leaks [A] | ✗ | | | | |
| | Unprotected invitation [A][D] | | ✗ | | | |
| | Missing anti-CSRF token [B] | | ✗ | | | |
| | Session hijacking [C] | | ✗ | ✗ | ✗ | |
| | Account bruteforce [F] | ✗ | | | | |
| Shopping for free | Account info tampering [A] | | ✗ | | | |
| | Price tampering [A] | | | ✗ | | |
| | Refund abuse [A] | | | | ✗ | |
| | Coupon leaks/tampering [A] | | | | | ✗ |
| Store defacement | Storefront tampering [A] | | | ✗ | | |
| Sensitive info disclosure | Unauthorized read requests [A] | ✗ | ✗ | | ✗ | |
| | Unprotected server database [C] | | ✗ | | | |
| Free membership | Sensitive parameter tampering [A][E] | | ✗ | | ✗ | |
| Denial of service | GraphQL cyclic queries [D]* | ✗ | ✗ | ✗ | ✗ | ✗ |

Table 4: Vulnerable patterns in platform operations that result into major (the first four) and relatively minor (the last two) attacks. [A]: Broken Access Control; [B]: CSRF; [C]: Improper Input Validation; [D]: Insecure GraphQL; [D]*: Tested on introspection queries; [E]: Mass Assignment; [F]: Weak Rate-limiting.

Table 4.

### 5.1.1 Full store takeover

12/32 platforms are susceptible to this attack (see Table 5) through 5 common vulnerable patterns (see Table 4). Out of these 5 patterns, *OTP/token leaks* and *unprotected invitation* are related to unauthorized access control and we discuss them here. Once logged into the victim's store as a merchant, an attacker can view customer and merchant users' data e.g., email, physical address, mobile number; de-link the victim merchant's bank account, and link their own account to withdraw the remaining order amount; and deactivate the store altogether. (The testing is done on our own accounts).

**OTP/token leaks.** The exposure of authentication tokens and OTPs (of merchants) in API responses enabled full store takeover in 3 platforms—*McaStore, Okshop* and *Lovelocal*. We could use the leaked session tokens to successfully login into the victim's account (using [10]). For example, in *Morecustomersapp* (50,000+ app installations), the server

| **Platform** | **Store takeover** | **Shopping for free** | **Store defacement** | **Sensitive info disclosure** | **Denial of service** | **Free membership** |
|---|---|---|---|---|---|---|
| Bikry | ●⋆ | ● | | ● | | |
| Branchbob | ❍⋆ | ❍ | ● | ❍ | | |
| Crystallize | ❍ | | | ❍ | ● | |
| GraphCMS | | | | ❍ | | |
| Lovelocal | ● | | | ❍ | | |
| McaStore | ● | | | ● | | |
| Mozello | ●⋆ | | | | | |
| Okshop | ● | ● | ● | ● | | |
| Shoppiko | ● | ● | ● | ● | | |
| Shopware | | | ● | | | |
| Storehippo | ● | ● | ● | ● | | ● |
| Swell | ❍ | | | | | |
| Wix | | | | | ● | |
| AbanteCart | ●⋆ | | | ❍ | | |
| Microweber | ❍⋆ | | | | | |
| nopCommerce | | ● | | | | |
| Saleor | | | | | ● | |
| WooGraphql | | | | ● | | |

Table 5: Overall results for security vulnerabilities in the tested platforms (with at least one major attack); ●: *platform-wide attacker* and ❍: *store-specific attacker* (⋆ implies victim interaction is needed); blank: not vulnerable. In instances where there are multiple attacker types, we consider the worst one (e.g., *platform-wide* attackers are worse than *store-specific* attackers), with the broadest scope; the last five platforms are open source.

verifies an authentication request[9] correctly, but there is no connection between this verification result and the generation of an authentication token; i.e., the token is sent to the client irrespective of the authentication result (fail/success). An attacker can simply enter *any* password for a target account (victim's email address), and receive the victim's authentication token, leading to full store takeover. The attacker needs to use victim's email address for a target attack, or simply a (large) list of known user email addresses for a platform-wide attack.

**Unprotected invitation.** From the profile management UI, a store merchant can invite new users to her store and assign them a desired role. We found that this functionality is insufficiently protected on *Storehippo, Shoppiko* and *Swell*. For instance, in *Storehippo*, there is an API endpoint to add merchant user into a store with a defined role. A low-privilege user role cannot make this API call, but *any* other merchant can call it to add merchant users into a victim's store. In *Swell*, only a low-privilege user could make such calls, and hence, only a *store-specific* attacker can exploit this vulnerability. In *Shoppiko*, the `Origin` URL is not attached to the session cookies present inside the request headers, allowing an attacker to change it to the victim's store URL, and thus update the user roles on the victim's store.

### 5.1.2 Shopping for free

6/32 platforms can be exploited to shop for free or lesser amount (see Table 5), through four vulnerable patterns—*account information tampering*, *product price modification, refund*

[9] /siteadminapp/api/SiteAdminSecurity/Login

*approval* and *coupon leaks/tampering*.

**Account information tampering.** By unauthorized modification of account details, e.g., the customer wallet balance and physical address, an attacker can shop for free, as found in the case of *Storehippo* and *nopCommerce*, respectively. In *Storehippo*, a malicious customer could target the profile management operation for any merchant to assign arbitrary balance to their own wallet and purchase store items. In *nopCommerce*, we found a redundant `address ID` parameter in the modify address functionality–one in the URL (which is validated by the server), and another in the request body (not validated). An attacker can exploit this by supplying her own `address ID` in the URL, and the victim customer's `address ID` in the request body. Note that the `address ID` parameter is 2-3 digit long and can be easily enumerated. The server's validation will be successful (as it is only checking in the URL) and the victim's address will be updated. An attacker can automate this to modify all customers' addresses in a store. A vigilant customer may notice their shipping address before finalizing an order; otherwise, the items will be shipped to the attacker.

**Product price modification.** We do not modify the product's price in a typical check-out process, rather we show another e-commerce operation (store management) in which an attacker can modify the price to shop for free. For example, in *Okshop* (500k+ installations), an attacker can generate a product modification request with the victim's store ID and any desired product price, including, *zero*, from her store dashboard. This request returns success, as *Okshop* only checks the existence of *any* authentication token (i.e., not necessarily the target's token) from a merchant role. The attacker can now go to the victim's storefront and authenticate as a customer to place an order for the tampered product for free.

**Refund approval.** E-commerce platforms often offer the functionality to give refunds. However, the refund request must be approved by the store merchant. An access control vulnerability would allow the attacker to approve her owns refund request, resulting in shopping for free. For instance, in *Storehippo* and *Okshop*, any malicious customer can trigger a "approve refund" API call, which has no access control (should be called only by the merchant user from their store dashboard). An attacker can exploit this by first filling the refund form (for her own order), and then triggering the process refund API call (supplying her own order ID).

**Coupon leaks/tampering.** Exposure of undisclosed coupon codes can result in shopping for free or a lesser price. For instance, in *WooGraphql* (v0.11.0, the latest release as of June 2022), which is a popular extension for *Woocommerce*, we found a missing access control check on a GraphQL query (used for fetching the coupon codes). Given a valid coupon ID, any unauthenticated attacker can make a GraphQL call to disclose the corresponding coupon code and the associated amount. The coupon ID for *any* shop in *WooGraphql* is a Base64 encoded value of a 2-3 digit integer. The attacker can enumerate all existing coupon codes by brute-forcing the coupon IDs on the vulnerable GraphQL query. An attacker can apply these coupon codes on her cart to reduce a product's price. Similarly, in *Branchbob*, each added coupon code is assigned a *coupon_id*, which is a 32 character long UUID value; the stores distribute only the coupon codes to their customers, and not the UUID values. However, the UUID is leaked when a customer applies a given coupon code into the cart. A malicious customer can then make an API call to modify the coupon's value and apply it to their cart to shop for free. We note that this issue can only be exploited by a *store-specific* attacker, as she would need a valid coupon code for a particular store.

### 5.1.3 Store defacement

5/32 platforms are vulnerable to unauthorized store defacement attacks through storefront tampering.

**Storefront tampering.** Unprotected storefront management interface allows storefront tampering in *Storehippo, Shoppiko, Shopware* and *Branchbob*. In *Storehippo* and *Shoppiko*, a merchant attacker can add *any* malicious HTML code, such as a form to collect plain-text customer credentials from the victim's store. An attacker can also add random products given a store ID. In *Branchbob*, the store dashboard allows merchants to create new pages inside the store, which requires proper authentication; however, existing pages remain unprotected against modification attacks. The API call to edit a page includes three parameters: store ID, page ID, content. We found an API call (can be called by any unauthenticated user) disclosing the store ID and every existing page's ID in the HTTP response. We can then add images, videos, texts, any HTML code, along with forms and JavaScript code, e.g., to collect customer data; we can also modify the privacy policy, and terms and conditions. In *Shopware*, an attacker can simply change the default store language to anything (which could potentially cause DoS). *Okshop* does not provide extensive functionality to customize or add HTML code, but allows an attacker to add random products on the victim's store, including illegal products such as drugs or guns, to defame the victim.

### 5.1.4 Sensitive information disclosure

11/32 platforms disclose sensitive information due to unprotected user information; see Table 5 for an overview.

**Unprotected user information.** API calls returning sensitive information about the existing store users or customers, such as email address, mobile number, name must be protected against unauthorized access. In *Storehippo*, we found an API disclosing user details such as the name, email and role of all merchant users. First, the request does not have sufficient access control checks and an attacker with a merchant role (on her own store) could make this API call for a victim's store. Second, upon further inspection (from API documentation), we found the parameter `fields` can be used to display selected profile items; e.g., if the `fields` is set to `[email,role]`, it would only display these two items in the response. We supplied a blank array in the `fields` value and the response from server contained all merchant users' data (for a single store), including: email, password (hashed), address, mobile number, OTP, activation code, role, and IP address. Note that the tests are conducted on our created stores only. An attacker can gain access to *any* store merchant's data by supplying the victim's store ID.

In *Crystallize*, a low-privileged store user can see name, email, and role of all the invited as well as existing store users due to an access control vulnerability. Similarly, in *GraphCMS*, an attacker can enumerate existing store users, given their email address. In *Okshop*, the list of potential *leads* (users who contact store merchants privately) in a store is disclosed (names and mobile numbers)–only the victim's public store ID is needed.

## 5.2 Insecure GraphQL Batching

We found that 8/32 platforms, namely *Shopify, Crystallize, BigCommerce, GraphCMS, Saleor, Wix, Magento*, and *WooGraphql*, make a call to standard GraphQL endpoints. In *Crystallize* and *Wix*, we found nested queries which were referencing to one another; as a result an attacker can create a malicious query, nested thousand-level deep to cause denial of service; for a 25-level nested query, the server took 18 seconds to generate a 68MB HTTP response. We did not attempt anything further as the platform is live. Also, in *Crystallize* we found that batching queries were enabled by default. Upon exploring the available introspection schema (documentation), we found a mutation used to redeem the merchant invitation token. With batching queries enabled, an attacker can guess the value of invitation tokens sent out by a victim merchant, by associating thousands of aliases with the redeem mutation, and then triggering a single GraphQL call. If an invite token is found, an attacker can successfully impersonate a low-privileged store user. We note that the length of invitation token is of 16 characters, which makes the attack less practical. Nevertheless, platforms should not rely upon the entropy of tokens alone, and should disable batching for such sensitive operations.

We found a similar issue in *GraphCMS*, where batching queries were enabled by default, but disabled for sensitive operations e.g., invite tokens. In *Saleor*, we found a defensive mechanism when executing nested and circular queries. Particularly, a query cost analysis mechanism is implemented which only allows query below a certain threshold to execute. However, the cost is not calculated for introspection queries (the queries for fetching the available documentation), which can be easily exploited for DoS. *Shopify* and *BigCommerce* appear to implement a secure query complexity analysis for preventing such attacks.

## 5.3 Injection Attacks

9/32 e-commerce platforms fail to properly validate the supplied inputs, and are vulnerable to injection attacks described in Sec. 3.3. In a popular open source e-commerce platform *AbanteCart* (v1.3.2, the latest release as of June 2022), we found that the update user page is vulnerable to reflected XSS via the user_id parameter. Although most of the HTML characters were properly sanitized, we found some unescaped HTML characters – e.g., `' { } ( )` – are reflected in the response. Since the values were reflected inside JavaScript code, we could use the unsanitized characters to execute our own JS code. An attacker could input malicious JavaScript code inside the user_id parameter and send the URL to any authenticated user; the JS code can be used to steal victim's session tokens resulting in account takeover. In *Bikry*, we found that a simple XSS payload injected in product reviews (as a customer), was reflected as it is in the merchant's store dashboard. An attacker could exploit this vulnerability by first registering as a customer in the victim's store and then using the product review functionality to drop a malicious JS payload inside reviews, which would execute on the victim's store dashboard. In *Microweber, Branchbob* and *Mozello*, we found a stored-XSS in the store settings functionality (accessible only to a merchant user), and hence this issue can only be exploited by a *store-specific* attacker. In *Sellfy* and

*Swell*, we found webhooks that trigger upon certain events, e.g., upon receiving an order, or receiving a contact form. An attacker can exploit this functionality by giving a local IP address with a random port as an input to the webhook and then trigger the predefined event. Based on the response error messages, an attacker can enumerate the open ports on the internal network of an e-commerce platform.

## 5.4 Mass Assignment

2/32 platforms are vulnerable to mass assignment, namely *Storehippo* and *Bikry*. In *Storehippo*, when a business owner wants to create an online store, she follows the normal merchant store creation process, and gets a store for a 14-day free trial period; a subscription fee must be paid afterwards. We found a mass assignment bug in the "store settings" functionality, allowing an attacker to extend the free trial period indefinitely (worth up to 400 USD/month). First, an attacker with a merchant role can view all the store setting parameters. Ideally, most of these parameters are assigned a value upon the merchant account creation phase and should not be modifiable. We observed a parameter *remaining_days*, which stores the number of days left for the subscription. This parameter is supposed to be controlled only by the SaaS platform, but an attacker can send a modified request message by altering this parameter, allowing a free subscription beyond the trial period. We confirmed this vulnerability by extending the trial period to 20 days, and our test store remained operational beyond the 14-day trial period.

We found another issue related to a product's price modification in *Bikry*. In particular, while adding a product to cart, a malicious customer can inject the *price* parameter while supplying any arbitrary value or even zero. The product would be added to cart with the customer chosen price, leading to financial losses for the merchant. We confirmed the vulnerability in our own store, by placing an order as a customer for a lesser price, and then checking our merchant dashboard for the order status.

## 5.5 Cross-Site Request Forgery

We identify missing anti-CSRF tokens as a vulnerable pattern, and if present in a sensitive e-commerce operation, this vulnerability might lead to store takeover attack (see Table 4). 5/32 platforms are vulnerable to CSRF. We note that a typical CSRF attack requires victim interaction. In *Mozello*, no anti-CSRF tokens are present in the merchant's store settings flow; an attacker can craft a form for settings modification (including the associated email) and may trick the victim to submit the form, thereby taking over the merchant's store. Similarly, *Bikry* is vulnerable to site-wide CSRF, i.e., no anti-CSRF tokens in any of the captured requests. Moreover, some state changing operations (e.g., adding a lower user role, enabling/disabling a coupon) are performed via a GET method. An attacker can simply create a URL to add herself as a store user and trick the victim into clicking the URL. Thereafter, the attacker would be added as a low-privileged user (and can exploit access control vulnerabilities to escalate the privileges to store merchant). In *AbanteCart* and *Branchbob*, the anti-CSRF token is present everywhere in the store dashboard, but not in their storefront. That is, no CSRF protection is present in “add to cart” or “add to wish-list” functionalities, allowing an attacker to add random products in the victim’s cart. The attacker would need valid product IDs (which are public) for this attack.

## 5.6 Weak Rate-limiting

9/32 platforms have a weak rate-limiting protection, as observed from our limited rate-limiting tests (50 consecutive attempts per platform, only on sensitive APIs, such as login or OTP verification APIs). This weakness can result into full account takeover, especially in the case of a 4-6 digit OTP, and denial of service attacks. *Lovelocal, Mcastore* implements a 4 digit OTP on signing-in and forget password functionalities. An attacker can brute-force all 10,000 combinations to takeover a store. *BigCartel, Bikry, Shoppiko* implement a 6 digit OTP, but does not block a brute-force attempt. The attacker can also spam text messages (OTP) on a victim’s mobile number using the platform’s login API.

## 5.7 Programming Stack of Open Source Platforms

Table 6 shows the technology stack used by the developers to build open source platforms that we analyzed. Based on our results, we did not find any correlation between the type of vulnerability discovered by our analysis framework and the programming framework used.

| Platform | Programming Stack | Vulnerable Patterns | Affected Functionality |
|---|---|---|---|
| AbanteCart | PHP, JS, Smarty | Session Hijacking, Unprotected Server DB | Profile Management |
| Magento | PHP, JS | N/A | N/A |
| Microweber | PHP, JS | Session Hijacking | Profile Management |
| nopCommerce | C#, ASP.net, TSQL, JS | Account info tampering | Profile Management |
| OpenCart | PHP, Twig, JS | N/A | N/A |
| Prestashop | PHP, Twig, TS/JS | N/A | N/A |
| Saleor | Python, Django, React | GraphQL Cyclic queries | All operations |
| Shopware | Symfony, Vue, PHP, Twig, TS/JS | N/A | N/A |
| Sylius | Symfony, PHP, Twig | N/A | N/A |
| WooGraphql | PHP | Coupon leaks/tampering | Coupon Handling |

Table 6: Programming stack of open source platforms

## 5.8 Performance Evaluation

We evaluate the performance of the automated part of our framework (broken authentication and access control) in terms of efficiency, and alerts removed by the filters.

**Efficiency.** The time required for generating an application model depends on the time taken by the tester to navigate through the application. The session modification component of our framework acts as a proxy, and the HTTP requests for a target e-commerce platform are recorded during navigation in form of a flow file. In order to create flow files for users with lower roles, we replay the saved requests with their session tokens. This replay is done using the MITM proxy's client replay feature which takes around 1 second to replay 2 requests on an average. We also reduce the flagged alerts with the help of filters which reduces the number of URLs to analyze manually. Thus, we improve the execution time by (1) running Session modification in parallel with the user's browsing session, and (2) removing alerts by filters.

| Vulnerable Platforms | Captured Unique Requests | Flagged Alerts | Alerts Removed by Filters | | | **Exploitable Alerts** |
|---|---|---|---|---|---|---|
| | | | Least-role | Option-method | Static-URL | |
| Storehippo | 254 | 127 | 34 | 0 | 9 | **84** |
| Swell | 219 | 182 | 62 | 0 | 0 | **120** |
| Branchbob | 181 | 72 | 4 | 35 | 0 | **33** |
| Shoppiko | 594 | 515 | 52 | 236 | 15 | **212** |
| Okshop | 53 | 24 | 8 | 0 | 9 | **7** |
| McaStore | 196 | 19 | 4 | 0 | 3 | **12** |
| Lovelocal | 183 | 52 | 9 | 15 | 24 | **4** |
| Bikry | 49 | 43 | 9 | 0 | 31 | **3** |

Table 7: The number of alerts automatically generated, filtered, and finally, the remaining exploitable alerts—for each vulnerable platform with at least one access control vulnerability.

**Removing alerts by filters.** The flagged alerts might contain false positives due to static URLs, API endpoints with "OPTIONS" method, and when detecting vertical privilege escalations. We reduce the number of false alerts, by identifying and eliminating alerts matching the defined filters in Sec. 3.1. We summarize the alert statistics Table 7; note that the "Captured Unique Requests" column represents the number of unique HTTP requests captured by our framework while browsing the platform as a merchant; "Flagged Alerts" are the number of requests flagged as possibly vulnerable; "Alerts Removed by Filters" are the number of requests removed by our filters; and finally, "Exploitable Alerts" are the number of true positives that represents the exploitable vulnerabilities (manually analyzed to understand their impacts). The total number of flagged alerts without applying any filter are 1034 (for the 8 vulnerable platforms). We note that we are able to reduce a total of 559 alerts after applying the three filters; i.e., 54.06% endpoints are removed automatically by our filter. Removal of these many endpoints helps us to apply our manual analysis only on the exploitable alerts to assess the impact of the vulnerabilities. We note that there can be some false negatives which are not identified by our framework; see Sec. 6.1.

# Chapter 6

# Discussion

In this section, we discuss our limitations, manual efforts, key takeaways, and recommendations for platform owners and merchants.

## 6.1 Limitations

1. Our detection of injection attacks, and mass assignment vulnerability is fully manual (for SaaS platforms) due to ethical reasons (automated in our local setup for open source solutions).

2. We do not consider access control bypass that requires too specific application-dependant payloads (e.g., parameter pollution payloads [5]).

3. Since we pre-define the list of keywords to detect the API requests that generate error upon multiple calls, we cannot determine new keywords on the fly. However, the list can be easily extended to include more relevant keywords.

4. We do not consider finding flaws in the payment systems or the checkout process as previous work extensively covered them (see Sec. 2.4). Moreover, we do not analyze other vulnerabilities from the OWASP testing guide, such as remote code execution (very few operations in e-commerce that can lead to remote code execution), server side template injection, DOM XSS which might be considered in the future work.

5. We evaluate platforms that offer a free/trial version for the merchants, not the paid-only platforms or the ones that require merchant/business identity verification (e.g., Taobao). Such platforms may pose new challenges, and have other categories of vulnerabilities (e.g., in ID verification steps). Even though these platforms might have a merchant vetting system, most of the attacks shown in our work can be conducted without requiring a merchant account.

**Manual efforts.** First, we manually browse the platforms as a store merchant while observing all the available roles. Since merchant is the highest role, browsing with it helps to cover more API URLs and generate a proper baseline request/response in the flow file. While manually browsing the store, we also need to observe available roles, and the type of authentication offered by the platform, such as cookie-based or header-based authentication. Second, we collect the session cookies for each platform by logging into the store and by analyzing the *Set-Cookie* response header from the Chrome browser's network tab (via the inspect element functionality). The browsed traffic and the session cookies help us to generate a flow file (containing HTTP traffic) by running the session modification component as described in Sec. 3. Manual browsing can be automated, with the risk of missing some critical features (e.g., the ones under several layers of menu/UI items). For impact analysis of the raised alerts (always true-positives), manual validation is required.

## 6.2 Key Takeaways

Based on our systematic study of 32 representative e-commerce platforms, we have some interesting and important observations.

1. The operational functionalities in e-commerce beyond checkout and payment can also have similar (shopping for free) and in some cases even much more significant security consequences (platform-wide store takeover). E-commerce operations (beside checkout) that can cause shopping for free include: profile/storefront management, and order and coupon handling; see Table 4.

2. Based on our results, access control vulnerabilities are more common compared to other types, which are also easier (i.e., no victim interaction is needed) to exploit than e.g., injection attacks or CSRF. This is possibly due to the fact that injection attacks or a CSRF can generally be prevented by the underlying framework that the platforms are built on, and there are predefined functions for input validation that the developers can use. However, for implementing a secure access control, the developers have to understand the underlying business logic, available roles, and properly implement role-to-object access mapping. Furthermore, within improper access control, we find that the unauthorized read vulnerabilities are more prevalent than unauthorized write; although write vulnerabilities appear to be more serious, we observe that read flaws

can be dangerous as well (e.g., store takeover due to access token leakage). Also note that generally there are more read operations than write in a typical platform interface.

3. Some platforms rely on the confidentiality of object IDs in order to prevent access control issues. This is not an appropriate mechanism as the object IDs may be extracted from other API responses.

4. We observe that e-commerce platforms are progressively incorporating GraphQL, which has its own security risks such as batching attacks and descriptive errors. These new vulnerabilities can provide powerful attack vectors for brute-forcing and denial of service.

5. Lastly, based on Table 6, we can conclude that the vulnerabilities we found were mainly introduced by the lack of proper validation performed by developers, i.e., the vulnerabilities are not inherent to the underlying programming stack. However, this conclusion must be interpreted with care given the small sample size of the tested open source platforms.

## 6.3 Recommendations

The vulnerable SaaS e-commerce providers should immediately fix the identified issues. Despite platform-level affects, some providers are slow in their response (as evident from our disclosure experience). They should also repeatedly use security frameworks like the proposed one, especially, when new features, roles are added, or security measures are modified. Merchants can also take advantage of our framework to check the security posture of a platform before selecting one. Customers, on the other hand, cannot take any active measure to detect/fix these vulnerabilities. However, along with a merchant's reputation/rating, they may also take into consideration the reported security/privacy weaknesses (if any) of the merchant's platform.

## 6.4 Conclusion and Future Work

SaaS based e-commerce platforms are hosting an ever increasing number of stores, but security vulnerabilities can affect such store merchants and users alike, if proper protection mechanisms are not enforced. Our designed security evaluation appears to be effective in terms of finding new vulnerabilities and answering our research questions. We found operations beyond checkout/payment, e.g., profile/storefront management, and order/coupon processing, can be easily exploited to shop for free. Beyond shopping for free, we also found more serious issues such as platform-wide store takeover, store defacement, and information disclosure. Several of our findings are due to unauthorized read/write access control issues and the use of GraphQL (beside REST APIs). Overall, we hope our work helps the platform admins to improve their security, and inspire other researchers to devise more comprehensive framework for effective security evaluation.

For future work, the Standard Development Kit (SDK) libraries offered by the e-commerce platforms could be checked for security issues.

# Bibliography

[1] Cross site scripting (XSS). https://owasp.org/www-community/attacks/xss/.

[2] ACPM. Android Xposed module to bypass SSL certificate validation. https://github.com/ac-pm/SSLUnpinning_Xposed.

[3] Barn2.com. How many websites use WooCommerce? https://barn2.com/woocommerce-stats/.

[4] Stefano Calzavara, Mauro Conti, Riccardo Focardi, Alvise Rabitti, and Gabriele Tolomei. Mitch: A machine learning approach to the black-box detection of csrf vulnerabilities. In *IEEE European Symposium on Security and Privacy19*, 2019.

[5] L. Carettoni and S. Paola. HTTP parameter pollution. https://owasp.org/www-pdf-archive/AppsecEU09_CarettoniDiPaola_v0.8.pdf.

[6] Brian Dean. Shopify revenue and merchant statistics in 2022. https://backlinko.com/shopify-stores.

[7] Kostas Drakonakis, Sotiris Ioannidis, and Jason Polakis. The cookie hunter: Automated black-box auditing for web authentication and authorization flaws. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, November 2020.

[8] Dolev Farhi. GraphQL security auditor. https://github.com/dolevf/graphql-cop.

[9] Folio3.com. Growing popularity for SaaS e-commerce platforms. https://ecommerce.folio3.com/blog/ecommerce-saas-platforms/.

[10] Christophe Gagnier. Cookie-editor. https://cookie-editor.cgagnier.ca/.

[11] Genymotion.com. Android virtual devices for all your development & testing needs. https://www.genymotion.com/.

[12] Mohammad Ghasemisharif, Chris Kanich, and Jason Polakis. Towards automated auditing for account and session management flaws in single sign-on deployments. In *oakland-long*, 2022.

[13] GraphQL.org. GraphQL introspection schema. https://graphql.org/learn/introspection/.

[14] Imperva. The state of security within e-commerce 2021. https://www.imperva.com/resources/reports/TheState_ofSecurityWithin_eCommerce2021_report.pdf.

[15] Michael Isbitski. API6: 2019 mass assignment. https://salt.security/blog/api6-2019-mass-assignment.

[16] Binny Joseph. 5 reasons of growing popularity of SaaS e-commerce platforms. https://www.linkedin.com/pulse/5-reasons-growing-popularity-saas-ecommerce-platforms-binny-joseph/.

[17] P. Kapoor, R. Pagey, M. Mannan, and A. Youssef. Silver surfers on the tech wave: Privacy analysis of android apps for the elderly. In *18th EAI International Conference on Security and Privacy in Communication Networks (Securecomm 2022)*, Kansas City, USA, October 2022.

[18] Xhelal Likaj, Soheil Khodayari, and Giancarlo Pellegrino. Where we stand (or fall): An analysis of CSRF defenses in web frameworks. In *24th International Symposium on Research in Attacks, Intrusions and Defenses*, San Sebastian, Spain, October 2021.

[19] Jiadong Lou, Xu Yuan, and Ning Zhang. Messy states of wiring: Vulnerabilities in emerging personal payment systems. In *30th USENIX Security Symposium (USENIX Security 21)*, August 2021.

[20] Mitmproxy.org. MITMproxy: A free and open source interactive HTTPS proxy. Version 8.0.

[21] Justin Moore. AutoRepeater: Automated HTTP request repeating with Burp Suite. https://github.com/nccgroup/AutoRepeater.

[22] Collin Mulliner, William Robertson, and Engin Kirda. VirtualSwindle: An automated attack against in-app billing on Android. In *ACM ASIA CCS'14*, Kyoto, Japan, 2014.

[23] OWASP. API top 10 - 2019, . https://owasp.org/www-project-api-security/.

[24] OWASP. OWASP top 10: 2021, broken access control, . https://owasp.org/Top10/A01_2021-Broken_Access_Control/.

[25] OWASP. Cross-site request forgery - OWASP, . https://owasp.org/www-community/attacks/csrf.

[26] OWASP. Testing for cross-site request forgery, . https://owasp.org/www-project-web-security-testing-guide/latest/4-Web_Application_Security_Testing/06-Session_Management_Testing/05-Testing_for_Cross_Site_Request_Forgery.

[27] OWASP. Testing GraphQL, . https://owasp.org/www-project-web-security-testing-guide/latest/4-Web_Application_Security_Testing/12-API_Testing/01-Testing_GraphQL.

[28] OWASP. OWASP top 10: 2021, injection attacks, . https://owasp.org/Top10/A03_2021-Injection/.

[29] OWASP. Lack of resources and rate-limiting, . https://github.com/OWASP/API-Security/blob/master/2019/en/src/0xa4-lack-of-resources-and-rate-limiting.md.

[30] OWASP. Session fixation, . https://owasp.org/www-community/attacks/Session_fixation.

[31] R. Pagey, M. Mannan, and A. Youssef. All your shops are belong to us: Security weaknesses in e-commerce platform. In *Proceedings of the ACM Web Conference (TheWebConf'23)*, Austin, Texas, USA, May 2023.

[32] Payload Box. SQL injection cheatsheet. https://github.com/payloadbox/sql-injection-payload-list.

[33] G. Pellegrino and D. Balzarotti. Toward black-box detection of logic flaws in web applications. In *Network and Distributed System Security Symposium (NDSS'14)*, San Diego, CA, USA, February 2014.

[34] Simon Reinhart. Auth analyzer. https://github.com/PortSwigger/auth-analyzer.

[35] Edward Roberts. E-commerce-under-attack. https://www.optiv.com/insights/discover/blog/black-friday-cybersecurity-covid-ecommerce-under-attack.

[36] Armin Ronacher. Jinja: a fast, expressive, extensible templating engine. https://jinja.palletsprojects.com/en/3.0.x/.

[37] SecurityInnovation.com. Authmatrix: Testing authorization in web applications and web services. https://github.com/SecurityInnovation/AuthMatrix.

[38] Shopify.com. 2022 shopify's biggest year ever. News article (February 16, 2022). https://news.shopify.com/2021-was-shopifys-biggest-year-ever-2022-lets-go.

[39] Somdev Sangwan. XSStrike. https://github.com/s0md3v/XSStrike.

[40] Sqlmap.org. SQLMap. https://github.com/sqlmapproject/sqlmap.

[41] Storehippo.com. Storehippo user roles. https://help.storehippo.com/topic/user-roles.

[42] Avinash Sudhodanan, Alessandro Armando, Roberto Carbone, and Luca Compagna.

Attack patterns for black-box security testing of multi-party web applications. In *NDSS'16*, San Diego, CA, USA, February 2016.

[43] F. Sun, L.Xu, and Z.Su. Detecting logic vulnerabilities in e-commerce applications. In *Network and Distributed System Security Symposium (NDSS'14)*, San Diego, CA, USA, February 2014.

[44] Z. Sun, A. Oest, P. Zhang, Carlos Rubio-Medrano, Tiffany Bao, Ruoyu Wang, Ziming Zhao, Yan Shoshitaishvili, Adam Doupé, and Gail-Joon Ahn. Having your cake and eating it: An analysis of Concession-Abuse-as-a-Service. In *USENIX Security Symposium*, August 2021.

[45] R. Wang, S. Chen, X. Wang, and S. Qadeer. How to shop for free online–security analysis of cashier-as-a-service based web stores. In *2011 IEEE symposium on security and privacy*, 2011.

[46] Wikipedia. Comparison of shopping cart software. https://en.wikipedia.org/wiki/Comparison_of_shopping_cart_software.

[47] Luyi Xing, Yangyi Chen, XiaoFeng Wang, and Shuo Chen. InteGuard: Toward automatic protection of third-party web service integrations. In *NDSS'13*, San Diego, CA, USA, February 2013.

[48] Wenbo Yang, Yuanyuan Zhang, Juanru Li, Hui Liu, Qing Wang, Yueheng Zhang, and Dawu Gu. Show me the money! Finding flawed implementations of third-party in-app payment in Android apps. In *NDSS'17*, San Diego, CA, USA, February 2017.