All Your Shops Are Belong to Us: Security Weaknesses in E-commerce Platforms

Rohan Pagey, Mohammad Mannan, Amr Youssef
Concordia University
Montreal, Quebec, Canada
{r_pagey,mmannan,youssef}@ciise.concordia.ca

ABSTRACT

Software as a Service (SaaS) e-commerce platforms for merchants allow individual business owners to set up their online stores almost instantly. Prior work has shown that the checkout flows and payment integration of some e-commerce applications are vulnerable to logic bugs with serious financial consequences, e.g., allowing "shopping for free". Apart from checkout and payment integration, vulnerabilities in other e-commerce operations have remained largely unexplored, even though they can have far more serious consequences, e.g., enabling "store takeover". In this work, we design and implement a security evaluation framework to uncover security vulnerabilities in e-commerce operations beyond checkout/payment integration. We use this framework to analyze 32 representative e-commerce platforms, including web services of 24 commercial SaaS platforms and 15 associated Android apps, and 8 open source platforms; these platforms host over 10 million stores as approximated through Google dorks. We uncover several new 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 (affecting 41000+ stores) and 6 platforms vulnerable to shopping for free (affecting 19000+ stores, approximated via Google dorks on Oct. 8, 2022). We have responsibly disclosed the vulnerabilities to all affected parties, and requested four CVEs (three assigned, and one is pending review).

CCS CONCEPTS

• **Security and privacy** → **Web application security**.

KEYWORDS

SaaS E-commerce Vulnerabilities, Access Control, Web Security

ACM Reference Format:
Rohan Pagey, Mohammad Mannan, Amr Youssef . 2023. *All Your Shops Are Belong to Us*: Security Weaknesses in E-commerce Platforms. In *Proceedings of the ACM Web Conference 2023 (WWW '23), April 30-May 4, 2023, Austin, TX, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3543507.3583319

1 INTRODUCTION

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 [4, 24]. 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 [7, 11].

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 an order manager with access only to the orders functionality. Such complexity may result into serious security vulnerabilities, as evident from recent incidents [10, 22], exposing customer data and causing account takeover.

Prior academic studies have also revealed several security and privacy issues related to business logic in e-commerce content management systems (CMS) [20, 29]. Notably, Wang et al. [32] 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. Following [32], Pellegrino et al. [20] explored more test cases (generated/executed automatically) to detect 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. Moreover, GraphQL APIs (providing more powerful functionalities than traditional REST APIs) in e-commerce platforms have also not been analyzed yet as they are relatively new (stable release in 2018).

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, how are the platforms' primary operations affected by unauthorized read/write access control issues and the use of GraphQL?

To answer these questions, we first identify a list of common operations for SaaS e-commerce platforms, by analyzing the network traffic, exploring the available functionalities and documentation. We consider vulnerabilities from past work on e-commerce platforms (e.g., [20, 29, 32]), known web attacks (e.g., [28]), OWASP API top 10 [16], and GraphQL vulnerabilities [19]. Finally, we identify

13 vulnerable patterns, and 6 categories of security consequences regarding the common e-commerce operations beyond checkout.

We then 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 detect vulnerabilities by considering multiple types of API requests and GraphQL calls. We rely on automation to detect all the core vulnerabilities. We detect improper input validation and mass assignment manually in the SaaS platforms due to ethical concerns. We store all the traffic (original and tampered) in a flow file, and detect all the core vulnerabilities by automatically analyzing this file. Our approach to detect broken authentication and access control is inspired from an open source tool *Auth analyzer* [21]. We also support automatic detection of vulnerabilities in API requests pertaining to creation and deletion operations. While our chosen categories of core vulnerabilities can be applied in other web services, we customize the vulnerable patterns and attacks for e-commerce platforms.

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.

Contributions and notable findings.

(1) We develop an experimental framework[1] 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 functionalities (beyond checkout/payment) offered by leading e-commerce solutions—closed-source SaaS products (via their websites and Android apps), and open source solutions. Our framework incorporates traditional web vulnerabilities as applicable to these e-commerce operations, and evaluates 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 37 serious security vulnerabilities, 12 of which allow *platform-wide* (i.e., affecting all stores) full store account takeover and 6 flaws allow shopping for free. More seriously, in 5/12 platform-wide account takeover and 5/6 shopping for free attacks, the attacker requires to know only the target's public store URL. Overall, 18 platforms are vulnerable to at least one major attack, and fail to adequately preserve the security of store owners/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 (*nopCommerce, 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—19,428 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 issue in two days after our disclosure. We requested a CVE for this vulnerability.

(6) A vulnerable GraphQL query in *WooGraphql*, which is an extension for *Woocommerce* (estimated to be used by 5 million stores as of 2022 [2]), allows an attacker to collect all existing coupon codes of a store and use them to shop for free, or a lower 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 (by exploiting XSS), and dumping the back-end database (by exploiting SQLi). We are assigned two CVEs for these flaws.

Ethical considerations and responsible disclosure. The vulnerabilities are analyzed only on our own accounts. We refrain from accessing sensitive information of other users. We shared our findings with proof-of-concept attacks, security consequences, and guidelines on possible fixes with the security team/developers of the affected platforms. We have been gradually communicating several times with the platforms in the past 6 months as the vulnerabilities are found. *Sellfy*, *Storehippo*, *AbanteCart* and *nopCommerce* have fixed our reported issues. Four platforms are still investigating the issues: *Swell*, *McaStore*, *Lovelocal* and *WooGraphql*.

2 E-COMMERCE ENTITIES, THREAT MODEL

SaaS e-commerce entities. A *merchant* is the central/admin role in SaaS e-commerce stores. They can build and customize an online store and are responsible for managing merchant users, tracking payment status, recording order details, and shipping products to customers. A merchant account, if compromised, leads to a full store takeover. A *merchant user* role in a store (e.g., store manager, reseller, order manager) is added by a store's merchant to support various store operations. 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.

Threat model. 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., order IDs of a store (known to some low-privileged merchant-users), store coupon codes (known to some store users). Compared to the platform-wide attacker, a store-specific attacker is limited in scope (i.e., the consequences is limited to a store). We consider attacks

[1]https://github.com/rohanpagey/E-commerce-Security-Analysis-Framework

that can be launched remotely, with or without requiring any user-interaction; we also do not require any physical access to merchant devices 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 do not deter a motivated attacker, as the returns are usually 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 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 content 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 take into account the following two minor goals: (5) *free membership* to upgrade to a paid plan offered by the platform, without paying any fee; and (6) *denial of service* to exhaust the server's resources by exploiting a website vulnerability.

3 SECURITY ANALYSIS FRAMEWORK

The security issues considered are inspired by previous research in the e-commerce security [20, 28, 29, 32] and refined by us to cover more vulnerabilities and attacks, as applicable to various e-commerce operations. For this purpose, we systematically review academic literature in web security [3, 5, 9, 12], and multiple non-academic resources [1, 16, 17]. 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 the MITM proxy and 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 automatically evaluate the flow file for different security issues; see Fig. 1. For analyzing the open source software for merchants, we download and deploy them locally on our lab systems. All the vulnerabilities are detected automatically for the open source platforms.

3.1 Broken Authentication and Access Control

For detecting these issues, we perform a sequence of steps: (1) we create two merchant accounts as the victim and attacker; (2) we login as the attacker account and supply her authentication token to our session modification component; (3) we browse the store UI from a victim account to generate API requests which we proxy through the session modification component.

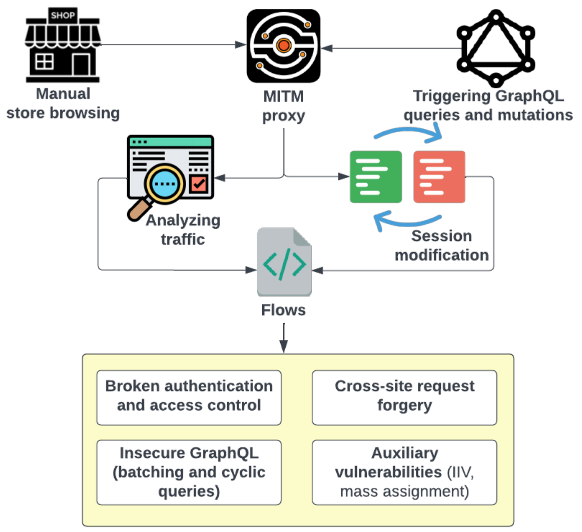
SHOP
Manual store browsing
MITM proxy
Triggering GraphQL queries and mutations
Analyzing traffic
Session modification
Flows
Broken authentication and access control
Cross-site request forgery
Insecure GraphQL (batching and cyclic queries)
Auxiliary vulnerabilities (IIV, mass assignment)

Figure 1: Overview of our proposed framework; all of the core vulnerabilities are analyzed automatically except auxiliary vulnerabilities, which are manual for SaaS platforms due to ethical concerns; IIV: improper input validation.

In order to analyze all API requests, the session modification component divides them into two types. (a) 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, the session modification component sends the original request to the server to get the expected response, and then forwards the modified request (by swapping merchant/attacker authentication tokens). Receiving the same response code and a similar content-length for both requests indicates an access control vulnerability. (b) 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 for all subsequent calls. Similarly, a POST request to register a user can only be called once (subsequent calls would generate a "user email already exists" error from the server). Requests to create an object also fall under this category, 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 the session modification component can determine an access control violation by only sending the modified request, and then checking the status code (e.g., 200 OK) and the response's content-length. To automatically classify the category of requests, the session modification component checks the HTTP method and the API endpoint of an incoming request. HTTP requests with a DELETE method always fall under type (b). Also, by manually exploring some platforms we populate a list of keywords (see Table 3 in the appendix) which are typically found in requests that fall under (b). POST/PUT API requests that contain these keywords (based on exact/partial matching) fall under type (b). All other requests are categorized as (a). Besides analyzing 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.

The flow files are also used as an input to other automated analysis components for detecting other core vulnerabilities (discussed in the following sections).

3.2 Cross-Site Request Forgery (CSRF)

A successful CSRF [3] attack allows an adversary to trigger state-changing requests (which can update some data on the server side–e.g., HTTP requests with POST, and PUT methods) in the victim's session. In case of a merchant account, a site-wide CSRF would allow an attacker to takeover all stores on the site. In order to perform a CSRF attack, 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 [18]): 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 (extracted from the flow file), we first check the content-type, and then search for anti-CSRF tokens in the 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. We create a list of regular expressions of such tokens based on *CSRF Scanner*[2]. State changing HTTP requests that do not contain anti-CSRF tokens are then flagged as vulnerable.

3.3 Insecure GraphQL

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 can 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. 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) [31] and brute-force attacks, especially in case of authenticating mutations.[3] 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. Moreover, 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 would reveal correct parameters required to make a successful call.

For analyzing an insecure GraphQL instance, we first detect if the platform under testing is using GraphQL. We do this by automatically searching the flow file for the presence of common GraphQL endpoints.[4] We also search on Google for any GraphQL documentation for the e-commerce platform under testing. Then, we use an open source GraphQL auditor tool [6] for detecting batching attacks and circular queries.

4 ANALYSIS RESULTS

We use our framework to analyze the security posture of 32 representative platforms, including 24 SaaS products (websites, and 15 Android apps for merchants/customers), and 8 open-source e-commerce platforms. 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 a search criterion. We also relied on Wikipedia [33] 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 and selected apps based on their installation/store counts, and availability of a trial plan (we also used the "similar apps" feature of the Play Store). Based on our results, we did not observe any correlation between the type of vulnerability discovered and the programming framework utilized by the platform.

In this section, we provide a summary of the findings and their impacts; for an overview, see Table 2. 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. While some of the attacks require the attacker to create their own shop, we found 5 vulnerable patterns—OTP/token leaks, missing anti-CSRF token, account info tampering, refund abuse, and coupon leaks/tampering—that can be exploited by an attacker without registering her own shop. These patterns can lead to serious attacks, e.g., store takeover and shopping for free (see Table 1).

4.1 Store Takeover

We found at-least one *store takeover* attack in 12/32 platforms through 4 common vulnerable patterns (see Table 1). 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; remove the victim merchant's bank account, and link their own account to withdraw the remaining order amount; and deactivate the store altogether. (The analysis was 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

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

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

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

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]		✗	✗	✗	
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]		✗		✗	
Denial of service	GraphQL cyclic queries [D]*	✗	✗	✗	✗	✗

Table 1: 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.

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 2: 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.

leaked session tokens to successfully login into the victim's account using a cookie editor [8]. For example, in *Morecustomersapp* (50,000+ app installations), the server verifies an authentication request 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 (failure/success), even though the UI shows a "login failed" message. 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 a 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. Also, in *Crystallize*, we found that GraphQL 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 invitation token is 16-character long, which may make the attack less practical as we do not know how many tokens can be tested in a single GraphQL call although there is no specified limit. Nevertheless, platforms should disable batching for such sensitive operations.

Missing anti-CSRF token. We identify missing anti-CSRF tokens as a vulnerable pattern, and if present in a sensitive e-commerce operation, this vulnerability would lead to store takeover attack (see Table 1). We found that 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.

Session hijacking. 5/32 e-commerce platforms fail to properly validate the supplied inputs, and are vulnerable to injection attacks described in Appendix A.2. In *AbanteCart*, 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 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 Javascript code can be used to steal victim's session tokens resulting in account takeover. In *Bikry*, we found that a 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 droping a malicious Javascript payload inside a product review, which would execute on the victim's store dashboard.

4.2 Shopping for free

6/32 platforms can be exploited to shop for free or lesser amount via four vulnerable patterns: *account information tampering*, *product price modification, refund 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 merchant can add some balance to a customer's wallet, which can be used by the customer while checking out a product. However, we found that the server does not verify the user calling this API. As a result, any malicious customer could target the profile management operation (designed for any merchant here) to assign arbitrary balance to their own wallet and purchase store items for free. 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 their 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.

Price tampering. We do not modify the product's price in a typical checkout process [29, 32], 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. In order to avoid any obvious suspicion, an attacker can also change the price to a *non-zero* value, such as a cent. We note that once the attacker modifies the product's price, a legitimate customer can also shop the affected product for a lesser price.

Refund approval. An access control vulnerability in the refund functionality would allow an attacker to approve her own refund requests, resulting in shopping for free. For instance, in *Storehippo* and *Okshop*, any malicious customer can trigger an "approve refund" API call, which has no access control (i.e., requires no merchant approval). 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) 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, anyone (without authentication) can make a GraphQL call to disclose the corresponding coupon code and the associated amount. The coupon ID for a particular shop in *WooGraphql* is a 3 digit integer (Base64 encoded). An attacker can enumerate all existing coupon codes by brute-forcing the coupon IDs via the vulnerable GraphQL query, and use the codes 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.

4.3 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, and 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. An attacker can then add images, videos, texts, any HTML code, along with forms and JavaScript code, e.g., to collect customer data; she can also modify the privacy policy, and terms and conditions.

4.4 Sensitive information disclosure

11/32 platforms disclose sensitive information due to unauthorized read requests, and one of them discloses due to an unprotected server database.

Unauthorized read requests. 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 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 item. 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 these vulnerabilities are evaluated 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 merchant user can see name, email, and role of all the invited as well as existing store users due to an access control vulnerability.

Unprotected server database. We found a time-based blind SQL injection in *AbanteCart's* customer update functionality (accessible to merchant). The SQLi allows a *store-specific* attacker to dump *AbanteCart's* database containing all the store's information such as the registered customer emails, passwords, order details, and so on. Although the information leaked is sensitive, an attacker would need a merchant role on a store to exploit this vulnerability (i.e., a store-specific attack). However, a malicious merchant can learn customer passwords which should not be available to anyone.

5 DISCUSSION

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

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).

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) 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. (3) We do not consider finding flaws in the payment systems or the checkout process as previous work extensively covered them (see Sec. 6). 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). (4) 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.

Lessons learned. 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 1. (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. (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) Lastly, 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.

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 RELATED WORK

Prior studies [20, 29, 32, 34] mainly focus on the security of the checkout process in an e-commerce storefront (visible to customers), and reveal numerous logic flaws when integrating services of third-party cashiers. Wang et al. [32] 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. [34] proposed a proxy model for testing stores that rely on third-party checkout integration and single sign-on authentication. Sun et al. [29] proposed a static analysis mechanism for detecting logic vulnerabilities in e-commerce applications while focusing on logic flow of the checkout process. In contrast, we focus on the merchant dashboard and other store management features, which revealed even more serious security flaws (e.g., store defacement, and platform-level attacks affecting all stores).

Sun et al. [30] analyzed 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. Xu et al. [35] discussed the emergence of an underground market called seller-reputation-escalation (SRE) markets, where online sellers can hire human labourers to conduct fake transactions in order to improve the reputation of their stores. Giancarlo and Davide [20] proposed 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 (see below), they also applied their framework on the checkout process in a customer-facing storefront.

Security concerns pertaining to the payment systems [13, 15, 36] in e-commerce have also been studied extensively in past research. Yang et al. [36] 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 2,679 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. [13] 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 [14, 21, 23] for detecting unauthorized read/write vulnerabilities in web applications. Generally, these tools replace session tokens or selected parameters in the original request, and then detect unauthorized access by comparing the differences between the modified and original responses. However, some tools [14, 23] do not handle the requests that generate error upon multiple calls. Auth-analyzer [21] is an improvement over the other tools as it supports all types of requests, although manual effort is needed for handling different request types. 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 3 in the appendix).

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 checkout process, even within a storefront (such as modifying customer details, adding coupon code). Our framework supports websites, Android apps, and open source products, since our core vulnerabilities do not involve creating any malformed operations as in [20]. Our analysis therefore sheds light on the broader picture of security vulnerabilities on e-commerce platforms. Since the logic vulnerabilities in checkout and 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 checkout process such as shopping for free. Our attack vectors would work even if the checkout process is securely implemented.

7 CONCLUSION

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 security evaluation framework is 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: platform-wide store takeover, store defacement, and large-scale information disclosure. Several of our findings are due to unauthorized read/write access control issues and the use of GraphQL. Overall, we hope our work can improve the platform security, and inspire other researchers to devise more comprehensive frameworks for effective security evaluation.

REFERENCES

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

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

[3] S. Calzavara, M. Conti, R. Focardi, A. Rabitti, and G. Tolomei. Mitch: A machine learning approach to the black-box detection of csrf vulnerabilities. In *IEEE European Symposium on Security and Privacy19*, 2019.

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

[5] K. Drakonakis, S. Ioannidis, and J. 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*, Nov. 2020.

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

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

[8] C. Gagnier. Cookie-editor. https://cookie-editor.cgagnier.ca/.

[9] M. Ghasemisharif, C. Kanich, and J. Polakis. Towards automated auditing for account and session management flaws in single sign-on deployments. In *IEEE Symposium on Security and Privacy*, 2022.

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

[11] B. 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/.

[12] X. Likaj, S. Khodayari, and G. 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, Oct. 2021.

[13] J. Lou, X. Yuan, and N. Zhang. Messy states of wiring: Vulnerabilities in emerging personal payment systems. In *30th USENIX Security Symposium (USENIX Security 21)*, Aug. 2021.

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

[15] C. Mulliner, W. Robertson, and E. Kirda. VirtualSwindle: An automated attack against in-app billing on Android. In *ACM ASIA CCS'14*, Kyoto, Japan, 2014.

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

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

[18] 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.

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

[20] 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, Feb. 2014.

[21] S. Reinhart. Auth analyzer. https://github.com/PortSwigger/auth-analyzer.

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

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

[24] 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.

[25] Somdev Sangwan. Arjun HTTP parameter discovery suite. https://github.com/s0md3v/Arjun.

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

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

[28] A. Sudhodanan, A. Armando, R. Carbone, and L. Compagna. Attack patterns for black-box security testing of multi-party web applications. In *NDSS'16*, San Diego, CA, USA, Feb. 2016.

[29] 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, Feb. 2014.

[30] Z. Sun, A. Oest, P. Zhang, C. Rubio-Medrano, T. Bao, R. Wang, Z. Zhao, Y. Shoshitaishvili, A. Doupé, and G.-J. Ahn. Having your cake and eating it: An analysis of Concession-Abuse-as-a-Service. In *USENIX Security Symposium*, Aug. 2021.

[31] Wallarm. GraphQL batching attacks. https://lab.wallarm.com/graphql-batching-attack/.

[32] 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.

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

[34] L. Xing, Y. Chen, X. Wang, and S. Chen. InteGuard: Toward automatic protection of third-party web service integrations. In *NDSS'13*, San Diego, CA, USA, Feb. 2013.

[35] H. Xu, D. Liu, H. Wang, and A. Stavrou. E-commerce reputation manipulation: The emergence of reputation-escalation-as-a-service. In *Proceedings of the 24th International Conference on World Wide Web*, Florence, Italy, May 2015.

[36] W. Yang, Y. Zhang, J. Li, H. Liu, Q. Wang, Y. Zhang, and D. Gu. Show me the money! Finding flawed implementations of third-party in-app payment in Android apps. In *NDSS'17*, San Diego, CA, USA, Feb. 2017.

A APPENDIX

A.1 Common E-commerce operations

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.

Store creation. A merchant requests a personalized store from a SaaS provider. If the request is successful, the store front/dashboard URL and a store ID is created and returned to the merchant.

Merchant and customer authentication. For both merchants and customers, authentication requires a user ID and password, which are verified by the CS. If successful, an authentication token is sent to the merchant/customer. The authentication request for the merchant originates from the store dashboard, while for customers it originates from the storefront.

Profile management. The operation allows a store merchant to modify store settings (e.g., adding new roles), and a customer to change their profile parameters (e.g., email).

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.

Order processing. This operation contains four sub-operations: order create, read, update, and delete. A customer can create an order request from a storefront URL, which is then verified by the merchant and CMS server. Once verified, the product is shipped.

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

A.2 Auxiliary vulnerabilities

Improper Input Validation. Essentially, in an injection attack, the attacker can send malicious input to the server which is then processed and may cause the application to behave in an unexpected way, such as running arbitrary JavaScript code in a victim's session (XSS) or injecting SQL queries to dump the application server's database (SQL injection). We first filter the saved flow file for selecting the URLs with GET method to analyze for XSS, and inputs the selected URLs to XSStrike [26], which outputs the vulnerable URLs. Then, for SQLi, we give the saved requests (from flow file) as an input to SQLMap [27], allowing us to determine if a request parameter is vulnerable or not. We avoid analyzing SQLi on SaaS platforms due to possible modifications on the server's database. For manual analysis of XSS on SaaS platforms, we inject a simple payload[5] 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 consider stored XSS primarily in the functionalities 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 a customer-injected payload executed in a merchant's session can allow the customer to access merchant dashboard and takeover the store.

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

Mass Assignment. A mass assignment vulnerability occurs when a server-side object's sensitive properties are inadequately protected. This can happen when 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`). For open source platforms, we use a parameter discovery tool [25] to find all the modifiable parameters in a given request. We assess mass assignment vulnerabilities manually (due to ethical reasons) for the SaaS platforms. Particularly, we first find a list of available properties by probing their API, checking the client-side code, or reading the API documentation. Second, we make an API request to modify the object, by including the known properties in the JSON request body. Third, before triggering the request, we change the property value according to its data type. After the modification request, we make a GET request to view the updated object property.

A.3 Minor attacks

Free Membership. *Storehippo* and *Bikry* are vulnerable to this attack via sensitive parameter tampering. In *Storehippo*, when a merchant gets a store for a 14-day free trial period; a subscription fee must be paid afterwards. We found a vulnerability in the "store settings" functionality allowing an attacker to extend the free trial period indefinitely. An attacker with a merchant role can modify the "remaining_days" (representing the store trial period) parameter of a store, allowing an indefinite free subscription. We confirmed this vulnerability by extending the trial period to 20 days, and our test store remained operational beyond the 14-day trial period. In *Bikry*, if a customer injects the price parameter with any arbitrary value or zero, they can get a product for a lower price or for free. This was confirmed by placing an order as a customer for a lower price, and then checking the merchant dashboard for the order.

Denial of Service. 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 that attackers could create malicious queries that were nested thousand-level deep, which can cause denial of service. For a 25-level nested query, the server took 18 seconds to generate a 68MB HTTP response. 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. We note that persistent-hashed-queries could help prevent DoS (e.g., quickly retrieving cached-query results when the same-query is made). However, we did not find any platform using such queries in their GraphQL schema.

A.4 Platforms

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 4 and 5.

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 3: 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.

Platform	# of stores (approx.)	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 4: 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 5: 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.