Understanding DevOps Security of Google Workspace Apps

Liuhuo Wan
University of Queensland
Australia

Chuan Yan
University of Queensland
Australia

Zicong Liu
University of Queensland
Australia

Haoyu Wang
Huazhong University of Science and Technology
China

Guangdong Bai*
City University of Hong Kong
China

Abstract

Cloud-based workspace systems, such as Google Workspace and Microsoft OneDrive, have enabled third-party developers to create and upload functionality-rich applications (referred to as *add-ons*). Existing studies have primarily examined user-centric data protection and permission management of this emerging ecosystem, but the underlying *DevOps* mechanisms that regulate add-on development, deployment, and operation remain largely unexplored.

In this work, we conduct the first *developer-centric* investigation of these DevOps mechanisms. We propose a hybrid method that combines a static analysis to abstract development and integration (i.e., deployment) (*Dev*) models and a dynamic analysis to add-ons' runtime operation workflows (*Ops*). It yields insights into the DevOps lifecycle of add-ons, unveiling associated attack surfaces and multiple types of security vulnerabilities, including source code leakage, code tampering and secret key exposure. Our large-scale evaluation of 5,300 Google Workspace add-ons reveals a concerning *status quo* of the ecosystem: 274 add-ons are subject to source code leakage, including widely-used ones with over 100,000 users. Among them, 96 (around one third) expose the secret keys of developers, e.g., PayPal merchant secret key and secret keys to access the developer's back-end databases.

CCS Concepts

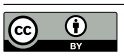
• **Security and privacy → Web application security**; • **Networks → Cloud computing**.

Keywords

Google Workspace, Application Security, DevOps Security, Reverse Engineering

ACM Reference Format:
Liuhuo Wan, Chuan Yan, Zicong Liu, Haoyu Wang, and Guangdong Bai. 2026. Understanding DevOps Security of Google Workspace Apps. In *2026 IEEE/ACM 48th International Conference on Software Engineering (ICSE '26), April 12–18, 2026, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3744916.3773245

* Corresponding author.



ICSE '26, Rio de Janeiro, Brazil

ACM ISBN 979-8-4007-2025-3/26/04
https://doi.org/10.1145/3744916.3773245

1 Introduction

Cloud-based workspace systems, such as Google Workspace [4] and Microsoft OneDrive [15], have become essential tools for individuals and organizations by enabling efficient data management and seamless team collaboration. They offer a cloud-hosted environment where users can store, access and share documents through browser-based features, without installing or updating software like traditional desktop applications (e.g., Microsoft Office 365). Instead, through the web browser alone, users can send messages (Google Chat), hold meetings (Google Meet), and collaborate on documents (Google Docs). This convenience has driven the widespread adoption of workspaces. Taking as an example Google Workspace, the leading workspace by user base, it has attracted over 3 billion active users [7].

To further expand workspace capabilities and meet diverse domain-specific user needs, workspace providers have introduced app marketplaces for third-party apps, known as *add-ons*. Third-party developers are enabled to create and publish add-ons that leverage the workspace APIs to provide specialized solutions. Examples of such add-ons include AI-enhanced tools for email composing [6], smart-home device managers [10], and code repository integrations [5]. They have greatly extended the core functionality of the workspace, leading to the rise of a workspace add-on ecosystem that continues to gain popularity.

Unlike mobile app development, which involves a heavy full-stack implementation, workspaces provide a lightweight *DevOps* environment for add-on developers. They expose easy-to-use APIs for the add-on developers to focus on add-on-related logic. For example, consider an add-on designed to upload user email attachments to the Box storage. By utilizing APIs, such as `GetEmailAttachment(ID)`, the add-on can easily fetch attachments and then send a web request to the Box service interface to complete the task. The development and deployment of add-on projects are cloud-based, and add-on developers can invite collaborators for *Dev* tasks. After deployment, add-on developers need to maintain and monitor interactions with service interfaces (e.g. the Box service interface) during *Ops* phase. While this lightweight *DevOps* design enhances convenience, it also introduces novel security attack surfaces for developers.

Recent research has extensively investigated security issues in add-ons, focusing on their permission control and data access. Regarding permissions, prior studies [34, 61, 62, 64, 68] utilize dynamic testing on add-ons to analyze permissions request and use, revealing that add-ons are subject to permission escalation and excessive permission requesting. In terms of user data protection, some studies examine potential data leakage in add-ons through

taint analysis [42, 63, 66, 70]. However, current research primarily focuses on *user-centric* security issues, with limited investigation into the add-on developer's perspective. The research question of *whether the confidentiality and integrity of developer assets (e.g., add-on source code, add-on project development/deployment and service interface interaction) are ensured throughout the entire DevOps lifecycle* has been largely open.

Our work. In this work, we conduct a comprehensive study of the *DevOps* (*development, deployment*, and *operations*) in Google Workspace (*workspace* for short hereafter) add-on ecosystem. Our work aims to explore three *developer-centric* research questions (RQs): *What are the DevOps models that developers follow to develop, deploy, and operate the add-ons (**RQ1**), what security issues can arise from such DevOps models (**RQ2**)*, and *what is the extent these security vulnerabilities affect real-world add-on projects that have passed workspace official vetting (**RQ3**).*

RQ1. Understanding DevOps models of add-ons. We adopt a hybrid analysis methodology, combining static and dynamic analysis, to understand the development, deployment and operation models of add-ons. A significant challenge is that the underlying *DevOps* of add-ons is a black box and lacks comprehensive documentation from the provider. To address this, we create our own test add-ons and conduct the testing on development, deployment and operation around them. Our exploration reveals that, each add-on is globally unique, identified by its *project ID*, and is under a *development* and *production* environment isolation mechanism. The add-on developers must handle or maintain the interaction with two types of service interfaces during *Ops* phase. Our analysis provides an in-depth understanding of the internals of add-on DevOps, laying the groundwork for our security assessment. We detail our analysis for RQ1 in **Section 3**.

RQ2. Investigating the security implications in add-ons' DevOps models. Based on the revealed DevOps mechanisms, we examine potential attack surface in add-on projects. We examine each phase a developer must go through and analyze whether an attacker can tamper with it based on its capabilities, considering the developer's code, permissions, and access to sensitive services. We successfully pinpoint three types of security issues associated with the DevOps cycle: the leak of add-on source code, the project development/deployment permission leakage, and secret key leakage, which allow access to sensitive resources such as developers' secret keys used to access databases or services (i.e., confidentiality) and unauthorized modification and deployment of add-on project (i.e., integrity). These vulnerabilities highlight the weaknesses present in the current add-on DevOps models. The component is detailed in **Section 4**.

RQ3. Studying the impact of security issues in real-world add-on projects. To assess the impact of these developer-centered security vulnerabilities in real-world add-on projects, we develop a detector to automate the detection process for the three types of leakage. The detector constructs links for various operations on the add-on code and the project, such as code revision and code deployment, using crucial parameters extracted from add-on network traffic. The links are then examined to check whether an attacker has the capability to conduct those operations without authorization. It also performs a taint analysis to identify secret keys used to interact with service interfaces. Our analysis of 5,300 official add-ons finds that 274 (5.2%) are subject to source code leakage. Among these 274 add-ons, 49 (17.9%) are also vulnerable to development/deployment permission leakage, and 96 (35.1%) are at risk of secret key leakage. RQ3 is detailed in **Section 5**.

Contributions. We summarize our contributions as follows:

- **Understanding the DevOps models of add-ons.** We investigate and thoroughly explain the DevOps models of workspace add-ons. From the *Dev* perspective, we illustrate how add-on projects are registered, developed, and deployed, along with their corresponding permission management. From the *Ops* perspective, we focus on how developers handle operational tasks, particularly managing service interface interactions in add-on projects.
- **A systematic security assessment and the practical impact.** We analyze attack surfaces arising from the underlying DevOps mechanisms, and have identified three types of vulnerabilities. The add-on code and project permission leakage can sabotage the confidentiality and integrity of the add-on project. The secret key leakage could grant an attacker's access to service interfaces essential for add-on developers, leading to data leakage and, in severe cases, unauthorized modifications to services provided by the service interface.
- **Revealing the *status quo* of the security issues of add-on *DevOps*.** Our findings indicate that the current *DevOps* of add-on is not sufficiently safeguarded. We provide mitigation strategies to help add-on developers address these issues. As the first study on the *DevOps* of add-on projects, our research not only contributes to improving existing projects but also sheds light on the future development of this ecosystem.

Ethics and Disclosure. To uphold ethical research practices, we have removed identifiable information from affected add-ons in our demonstrations and case studies. We have never used leaked secret keys to access data exposed by the add-on developers' service interface. All detected vulnerabilities were reported to Google Workspace through the Bug Hunter [18]. We also contacted the affected add-on developers via email. Both Google and the add-on developers acknowledged the identified vulnerabilities. Although Google does not disclose their mitigation for these issues, our recent visit has found that they have disabled certain features in the add-on *DevOps* process to address these vulnerabilities.

Roadmap. RQ1 (Section 3) uncovers the DevOps models through hybrid analysis. These models form the basis for identifying potential attack points in RQ2 (Section 4) and analyzing the discovered issues in RQ3 (Sections 5).

2 Background

Architecture from the developer side. In this section, we briefly introduce the architecture of workspace add-ons, to facilitate the understanding of our analysis. There are four parties involved in ensuring the normal functionality of add-on projects, which are **workspace client**, **workspace server**, **add-on front-end** and **add-on back-end**. Each party is responsible for different aspects of this ecosystem.

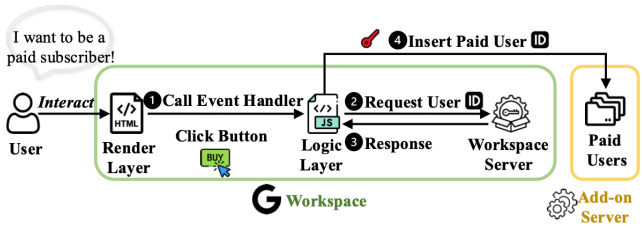


Figure 1: User interaction workflow of the add-on

- **Workspace Client (WC)** is the web application provided by workspace. WC is directly displayed to users and communicates with them in the end browser. In addition to native applications, it renders the HTML pages of add-ons.
- **Workspace Server (WS)** securely manages sensitive user resources and workspace-related services, such as authentication secret keys, data, and access permissions. It verifies user and add-on identities to ensure authorized access. Besides direct access from users, add-on developers can interact with it via exposed APIs. The WS handles requests from users and add-ons, executing actions such as data retrieval or modification.
- **Add-on Front-end (AF)** is running on top of workspace client. AF consists of two modules: the *render layer*, which includes HTML pages, and the *logic layer*, which contains JavaScript code. The render layer serves as the core to communicate with users. The logic layer would interact with the render layer through an event handler. In the current design, the logic layer is only visible to the WS, remains invisible to end users, and is not sent to the browser side. AF is the add-on project that developers must provide for workspace, and our work investigates the security risks inherent in the DevOps model of add-on projects.
- **Add-on Back-end (AB)** refers to the server or service owned and maintained by the developer. It interacts with both AF and WS through service interfaces, facilitating resource transfers for users and providing services to front-end users.

Interaction from the user side. We examine the communication between these four parties during the user's interaction with the add-on. As illustrated in Figure 1, when the user initiates the workspace (**WC**) and interacts with the front-end *render layer* (**AF**), specific trigger events (e.g., the purchase button is clicked, **Step ❶**) are transmitted to the *logic layer* via network requests. The logic layer (**AF**) then executes the corresponding function. During this process, it may communicate with the workspace server (**WS**) to access certain services (e.g., retrieving the current user's ID, **Step ❷** and **❸**). Additionally, the logic layer may interact with add-on's back-end (**AB**), such as inserting the user's paid license (**Step ❹**) into the add-on developer's database through the service interface. As shown in Figure 1, the add-on back-end is maintained on the add-on server rather than on the workspace server to ensure the confidentiality of both add-on data and service details.

3 Understanding DevOps Models of Add-ons (RQ1)

Due to the black-box nature of the workspace and add-ons, we resort to building a test add-on for our analysis, comprising HTML pages and a JavaScript file. Additionally, we set up an add-on server to handle data requests from its client.

3.1 *DevOps* exploration

***Dev* exploration.** The development environment is implemented as a web-based interface. To extract the *Dev* phase protocol, we develop a Selenium-based [35] web element detector to analyze, identify, and recognize those elements that the developer can interact with during the *Dev* phase. The detector extracts three types of elements [40] that encompass general scenarios: clickable buttons, editable text fields, and selectable options, by searching for element tags and attributes. For example, a button may appear as a `<button>` tag or a `<div>` with the attribute `role="button"`. To further understand the functionality of the identified elements, we interact with them directly. For editable text fields, we rely on manual input. This approach allows for more flexibility and precision in code-editing environments, which require advanced programming expertise for complex logic flow control and customization. To ensure efficiency, we use code snippets from the official template [30] provided by workspace for these editable text fields. Click actions are initiated for buttons, while each selectable option is chosen one by one.

We record the changes in the HTML pages of the web interface and network packets that occur before and after each operation (e.g., clicking a button), and then compare the differences. This differential analysis reveals that any operations (e.g., modification) in the add-on project do not take effect immediately until the *deploy* button is clicked. This way, we find that there is a *production* environment managed by the workspace, to which developers can push the developed add-on code.

***Ops* exploration.** We continue to explore how developers handle service connections and maintenance, which constitute the *Ops* phase. After successfully deploying the test add-on, we monitor both the web requests initiated from and received by the add-on for one week. This step enables us to extract all the services that add-on developers must interact with and monitor during the *Ops* phase. Next, we classify all captured web requests based on their domain names. Finally, we use Django's request handling mechanism to identify the interfaces responsible for processing these web requests, capturing both the interface name and the parameters passed to each interface. Through this systematic dynamic analysis, we identify two types of service interfaces (including the interface name and its corresponding parameters) that add-on developer need to interact or maintain : 1) service interfaces implemented by add-on developers, and 2) service interfaces implemented by workspace. Interaction details are given in Section 3.3.

3.2 Add-on *Dev* Phase

This section presents the uncovered *Dev* phase (Figure 2).

Add-on registration. All add-on projects are stored on a cloud platform provided by workspace and are referred to as *cloud projects* [13]. Therefore, the developer is required to register the add-on project first (**Step** ① in Figure 2). During registration, the add-on is assigned a globally unique project ID. For developers managing multiple add-ons, each add-on is assigned a unique project ID.

Add-on development & updating (development environment). After registration, the developer can build (**Step** ②) an add-on in their *development environment*. The development environment

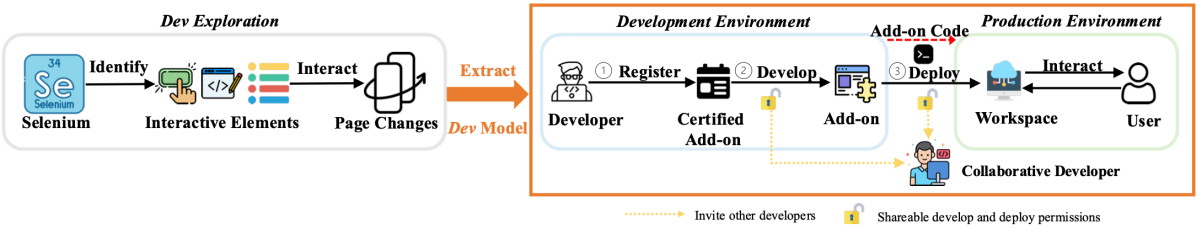


Figure 2: The *Dev* phase of the add-on

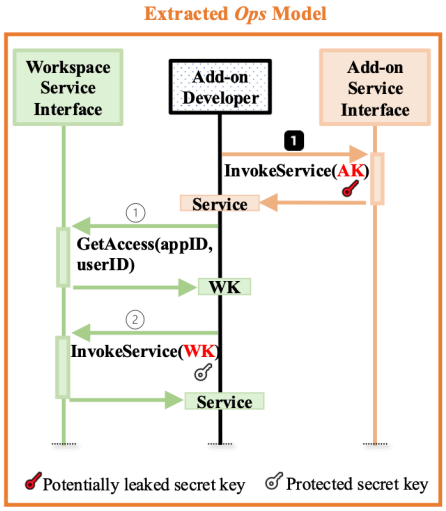


Figure 3: Service interfaces implemented by add-on and workspace developers

allows the developer to update their deployed add-on, such as adding features or fixing bugs. During this phase, users and cloud-based workspaces remain unaffected. Users continue to use the previously deployed add-on in the *production environment*.

Add-on deployment (production environment). Once the developer finishes developing or updating the add-on project, they can deploy the updated version. After deployment, the add-on code (AC) becomes available in the *production environment* (**Step** ③). Every time a user interacts with the add-on, workspace fetches and renders the newly deployed version from the *production environment*. If needed, the developer can also revert to a previous deployment, providing flexibility in managing updates and ensuring stability.

The registration must be completed by the add-on owner. However, the owner can invite collaborators (through *invite* button) with authorized access for development (DevPerm in **Step** ②) and deployment (DepPerm in **Step** ③) based on our exploration, as indicated by the unlock icon (🔓) in Figure 2.

3.3 Add-on *Ops* Phase

As identified during the *Ops exploration* (Section 3.1), there are two types of service interfaces that add-on developers must interact with and maintain. If these exposed service interfaces are not securely protected, they can be exploited by attackers, leading to potential security risks.

Service interfaces implemented by workspace developer: As shown by the green lines in Figure 3, for add-on developer to access such service interfaces, they queries with the *app id* and *user id* (**Step** ①) to retrieve the secret key (WK specified for the user). Then the add-on developer simply provides this WK (**Step** ②) to invoke the interface and access resources or services specified by the *user id*.

Service interfaces implemented by add-on developer: Access to the add-on service interfaces is controlled through permission management implemented by the add-on developer, and this can vary significantly depending on the customized implementation. We use a simple yet general workflow in Figure 3 to represent this process: when external services communicate with the add-on interface (orange lines), they must provide the add-on secret key (AK) for verification (**Step** 1).

4 Assessing Security Implications within Add-on DevOps (RQ2)

Based on the extracted DevOps model, we explore developer-centred attack surface against the critical assets of the developer based on a realistic threat model. We define a realistic threat model (Section 4.1), and then examine vulnerabilities and security implications in the add-on DevOps lifecycle (Section 4.2).

4.1 Threat Model

Adversary. To ensure that our threat model and attack scenarios are firmly grounded in established security research, we examine existing work on the security of multi-entity platforms or applications [27, 38, 42, 68, 70]. They commonly adopt a threat model involving web users who attempt to access confidential data or permissions beyond their authorized scope, like accessing other users' data. In our work, we adopt a similar threat model, assuming an adversary who has a valid workspace account and specifically targets *developers'* data or permissions. They can access the add-on by installing it in their workspace and interacting with it as a regular user. During this process, the adversary aims to obtain the add-on developer's confidential data [38, 70] (e.g., AC, WK, and AK) or permissions [42, 68] (e.g., DevPerm and DepPerm), which are summarized in Table 1 (column **Keys/Permissions**).

Adversarial capabilities. We assume that the adversary has complete control over their own workspace account and devices [27, 38], without losing practicality. This implies that the adversary can access and interact with any add-on available in the official marketplace. Additionally, the adversary can monitor and analyze all network traffic exchanged between workspace or add-on and their device [38, 42, 70, 71]. We make no unrealistic assumptions about developers' security awareness or capability, as noted by prior

Table 1: A summary of the attack feasibilities within the add-on DevOps

	Data/Permissions	Functionality	Target victim	Impact	Feasible
Add-on Dev Phase	-	Registration	Developer	*N.A.*	✗
	DevPerm	Development	Developer & User	Development tampering	✓
	DepPerm & AC	Deployment	Developer & User	Deployment tampering	✓
Add-on Ops Phase	WK: workspace secret key	Access to workspace service interfaces	User	*N.A.*	✗
	AK: add-on secret key	Access to add-on service interfaces	Developer & User	Data leakage/Service manipulation	✓

Table 2: Comparison between workspace studies and our work

Platform type	User-centric		Developer-centric	
	Permissions	Cross-application	*Dev* phase	*Ops* phase
Google Workspace	[55] [34]	[23] [54]	○	○
Microsoft OneDrive	*N.A.*	[25] [27]	●	○
Slack	[27]	[68] [27]	○	○

Prior work targets user-side vulnerabilities; we study developer-side ones. ● = no exposure, ○ = exposure.

work [49]. As our work focuses on the inherent security of DevOps, we assume that no social engineering is used. This means that the victims (i.e., add-on developers) are not persuaded by the adversary [71] to trivially disclose any confidential information. They are also not tricked into taking unsafe actions [27, 61], such as clicking maliciously crafted URLs or installing spyware, prior to launching the attacks presented in the paper. This assumption aligns with STRIDE [1], which separates technical attack surfaces from social engineering threats. Apart from interacting with the add-on as a *normal user*, the adversary has no further access to the add-on project or its service interfaces. In addition, we assume that the workspace servers (**WS**) are secure and do not collude with the adversary. Attacking such infrastructure is an orthogonal research direction.

4.2 Attack Scenarios

As our work is the first to focus on security issues that directly affect developers, we turn to a broader literature to guide our analysis. Through an in-depth analysis of online developer security guidelines, a previous study [19] identified several developer-centered security concerns, such as secret handling, improper permission delegation, and mismanagement of source code storage. Building on these concerns, we analyze the practicality [49] of attacking each functionality in workspace's DevOps process, covering add-on development, deployment, and related service interfaces. This helps us derive a comprehensive list of exploitable points, which are summarized in Table 1. Each exploitable point is detailed below.

***Dev* registration.**

Impractical. It is impractical to attack the add-on registration, as it requires the adversary to control the add-on admin [27, 28], which is beyond their attack capabilities.

***Dev* development.**

***Leakage 1: with the leak of* DevPerm.** The DevPerm is originally intended for authorized collaborators of the add-on project. Existing research has revealed that with leaking DevPerm allows the attacker to manipulate the add-on project [21, 74], such as by inserting a backdoor code snippet [47]. If the developer is unaware of this modification, they may unknowingly deploy a compromised version.

***Dev* deployment.**

***Leakage 2: with the leak of* AC.** Theoretically, the add-on code (AC) is inaccessible to anyone except the add-on developer and the workspace. The workspace requires access to the add-on project to facilitate rendering the add-on for users. However, the AC is not transmitted to the browser side [27, 54, 55, 68] and remains invisible to end-users. Prior studies have shown that any exposure of the AC may result in serious compromises to code confidentiality [65, 73].

***Leakage 3: with the leak of* DepPerm.** Similarly, if the DepPerm is leaked, the attacker can interfere with the **add-on deployment** stage. They can gain access to the add-on developers' credentials and use them to impersonate the developers and upload malicious code [45, 46, 50]. They can potentially deploy any version of the add-on, including reverting to a previous buggy version, as indicated by the name and description of the version, thereby disrupting its normal functionality.

A more severe case occurs if both the DevPerm and DepPerm are leaked, as the attacker would gain nearly full control of the add-on project. The DevPerm and DepPerm are bundled together in the current design of workspace. Furthermore, the leak of DevPerm and DepPerm also implies the leak of AC.

***Ops* service interface interaction.**

***Leakage 4: with the leak of* AK.** Developers are responsible for managing the AK. However, under AC leakage, attackers can gain insight into the service interfaces implemented by add-on developers, such as the interface for access databases or other services [42]. In such a situation, leaking the secret key AK [41, 42, 65, 67, 70] enables unauthorized access to or manipulation of the developer-maintained service through its interfaces [42, 70].

***Impractical: with the leak of* WK.** Since WK is associated with an individual user as shown in Figure 3, the workspace verifies the validity of the user ID when the add-on developers request the WK. Obtaining such a WK is difficult, as it requires the attacker to bypass the user identification verification [33, 70] enforced by the workspace, which is beyond the attacker's capabilities. Furthermore, the WK is updated with every query. Even if a valid WK is obtained, it can only be used once before it expires [27, 69]. Therefore, we do not consider the leakage of WK to be practical.

Comparison. We provide a comparison between our work and existing studies in Table 2. While existing studies primarily focus on user-centric security vulnerabilities, our work is the first to investigate vulnerabilities from a developer-centric perspective. We analyze the most popular cloud-based workspaces, which together account for over 90% of the market share [7, 14]: Google Workspace (73.02% [7]), Slack (24.31% [14]), and OneDrive (0.33% [7]). All cloud-based workspaces follow the general design discussed in Section 3.

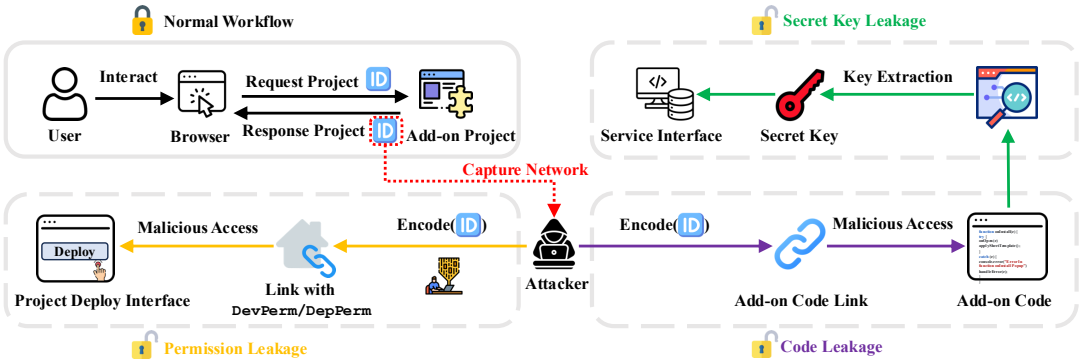


Figure 4: The attack flow of three leaks in add-on *DevOps* models. *Upper left*: normal workflow; *Upper right*: secret key leakage; *Lower left*: permission leakage; *Lower right*: code leakage.

However, Microsoft OneDrive demonstrates resilience against development and deployment attacks because it does not support collaboration, only a single role, the *admin* has control over the *Dev* phase. In such a case, the attacker would need to control the *admin*, which is beyond the scope of his attack capabilities. Despite this, it remains vulnerable to the remaining attacks. In contrast, Google Workspace and Slack are susceptible to all DevOps-related vulnerabilities.

5 Assessing the Prevalence of Leakage Vulnerabilities (RQ3)

The Three-Layer Assessment Model. In this section, we focus only on practical leaks (leakage 1 to 4). All of these exposures we have identified are based on the **Code Layer Leakage** (AC, detailed in Section 5.1), as the attacker obtains confidential code to which they should not have access. We begin our study by examining the leak of add-on code, then we further investigate whether the attacker has the capability to manipulate the add-on project, which is the **Permissions Layer Leakage** (DevPerm, DepPerm, detailed in Section 5.2). Lastly, building on the code leakage, we analyze whether the service interface is properly secured. If not, and the add-on developer unintentionally exposes service secret keys, it could lead to the **Service Layer Leakage**, detailed in Section 5.3. We present a case study that compares normal and malicious access scenarios to illustrate how these vulnerabilities are introduced and their severe impacts, following the measurement of their prevalence in real-world add-on projects that have passed vetting. Figure 4 illustrates the workflow of all leakage cases. We use dynamic analysis to extract project IDs from network traffic and generate links to access source code and permissions, then apply taint analysis to detect secret key leakage.

5.1 Assessment of Leak of Code

Normal Workflow. In a typical workflow, add-on projects allow collaboration only from invited collaborators, all within the project owner's awareness. Access to the add-on project is restricted from attackers.

Unauthorized Access and Attack. Figure 4 illustrates how the add-on code is leaked from the attacker's perspective. Specifically, the attacker interacts with an add-on in a user role, during which a network traffic packet containing a valid *project ID* is sent to the browser, as shown in the upper left part of Figure 4. The *project ID* plays an important role, as the storage address of AC can be restored by encoding it as a parameter into a URL. Finally, we encode this *project ID* into the formatted URL and attempt to access it as shown in the lower right part (purple lines) of Figure 4. In this process, we can get two types of responses: "*You are restricted from accessing this project*," and the normal response of the AC code. The second situation is marked as AC leakage, and the returned result is shown in Figure 4.

To estimate the prevalence of AC leakage, we extract the *project ID* for each add-on. Since the *project ID* is transferred only when the user interacts with the add-on, we built a Selenium-based [35] tool to automate this interaction. First, we enable the use of add-on from the marketplace by clicking the *install* button. Then, Selenium launches the workspace service, such as opening a Google Doc, and identifies the enabled add-on on the toolbar. The tool clicks the add-on to initiate the *open add-on* operation, during which network traffic is recorded. Finally, we *uninstall* the add-on and proceed to the next one. The *project ID* is straightforward to identify, as it is a fixed-length string. From the recorded network traffic, we extract all 57-character-long strings. We construct the AC address, then send web requests to verify their accessibility by unauthorized attackers.

Evaluation. This process resulted in the identification of 274 add-ons with AC leakage. The distribution of these add-ons is depicted in Table 3. In general, add-ons with a larger user base are less likely to experience AC leakage, with the percentage decreasing from 14.61% for add-ons with less than 100 users to 0.94% for add-ons with more than 100,000 users. However, it is concerning that there are still 11 add-ons with over 100,000 users that are vulnerable. We categorize the vulnerable add-ons by developer and find that while most developers own only one add-on, some have up to seven vulnerable add-ons. There is a noticeable tendency that developers managing multiple add-ons tend to leak all ACs.

We also measure the distribution of requested permissions for add-on with code leakage, as shown in Figure 5. The blue bars highlights add-ons with code leakage. The most frequently requested permission is *Email*, which appears 178 times in the leaked AC. Another sensitive permission, *Connect to external services*, is the second most requested. This indicates that the majority of vulnerable add-ons connect to and utilize external services. These vulnerable

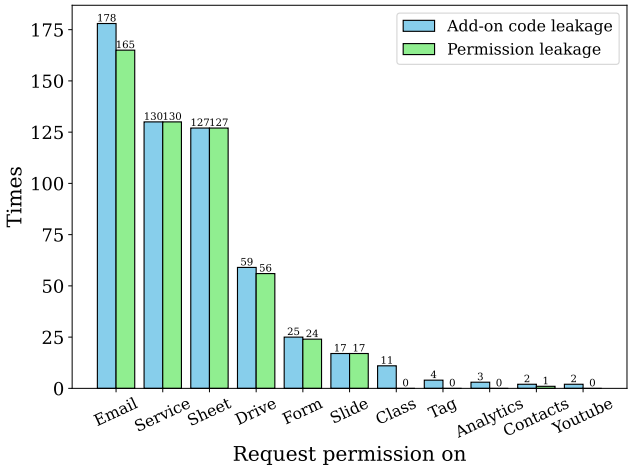


Figure 5: Request permissions in vulnerable add-ons

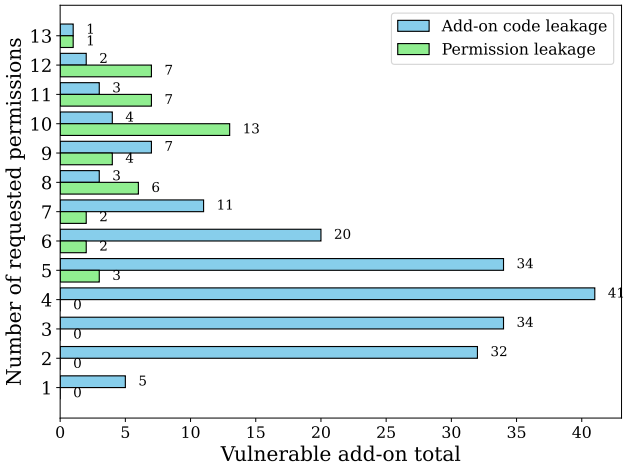


Figure 6: Permission counts in vulnerable add-ons

Table 3: The user base distribution of add-on with leaks

User base	100		100 - 1,000		1,000 - 10,000		10,000 - 100,000		100,000 +	
Total Add-ons	1006		797		1142		1182		1173	
	#Add-on	%Vuln	#Add-on	%Vuln	#Add-on	%Vuln	#Add-on	%Vuln	#Add-on	%Vuln
Add-on code	147	14.61	55	6.90	38	3.33	23	1.94	11	0.94
DevPerm& DepPerm	22	2.18	17	2.13	4	0.35	4	0.34	1	0.09
Secret key	53	5.26	20	2.51	15	1.31	7	0.59	1	0.09

add-ons primarily interact with workspace services such as email, spreadsheets, drives, forms, and slides. A smaller proportion are associated with classes, tags, analytics, and contacts. Attackers may further exploit these services to compromise user privacy through the affected add-ons. These services are either education-focused or marketing-focused. Add-ons requesting more user permissions pose greater security risks, as attackers can misuse them to exfiltrate user data. As shown in Figure 6 (blue bars), the y-axis indicates the number of requested permissions, while the x-axis shows how many add-ons request that number of permissions. The largest group of add-ons (a total of 41) requests 4 permissions, while some request as many as 13 permissions.

Case Study. One add-on, *Highlight*, with 141,317 users, last modified on July 29, 2024, leaks its `AC`. We successfully retrieved the source code of *Highlight*, gaining detailed knowledge of the add-on project, including the developer key, resource storage address, and the associated retrieval logic. The developer key, used as the developer identification to access the spreadsheets, was exposed in the source code. As demonstrated in previous studies [25], attackers can exploit this key and the standard API provided by Google to impersonating the add-on developer and steal sensitive resources.

5.2 Assessment of Leak of Permissions

Normal workflow. An attacker should not be able to obtain the `DevPerm` or `DepPerm` privileges to modify or deploy the add-on project.

Unauthorized Access and Attacks. Similarly, the link containing `DevPerm` or `DepPerm` can be restored by encoding *project ID* into the URL as shown in the lower left of Figure 4. We use a similar method (to that of code leakage) to verify if the attacker has gained `DevPerm` and `DepPerm`. For all constructed links, we sent web requests to verify whether the attacker is authorized to develop or deploy the add-on project. Two types of responses were returned: 1) "*Ask the project owner to share the development and deployment permissions with you*", and 2) No prompt message, but the interface to develop or deploy the add-on project is accessible as shown in the lower left part of Figure 4. We mark the second case as *DevPerm*/*DepPerm* leakage (yellow lines in Figure 4).

Evaluation. As shown in Table 3, 49 add-ons leak their `DevPerm` and `DepPerm`. This vulnerability persists even for add-ons with a large user base. As shown in Table 3, 2.13% of add-ons with fewer than 1,000 users are vulnerable, and approximately 0.35% of add-ons with 100,000 users remain vulnerable. Considering the severe impact on both developers and users, this prevalence is alarming. Interestingly, we find that add-ons vulnerable to the leak of `DevPerm` and `DepPerm` are often maintained frequently by their developers. Twelve were last modified in 2023, and fourteen were last modified in 2024. This suggests that `DevPerm` and `DepPerm` leakage are widespread, not only in legacy add-ons, but also in actively-maintained projects. A study on GitHub [41] also observed this phenomenon. Despite regular maintenance, certain security risks persist in projects. As illustrated in Figure 5, add-ons with permission leakage (green bars) request permissions on key services like *Email* (165 times), *Sheet* (130 times) and *Drive* (56 times). Furthermore, Figure 6 shows that add-ons leaking `DevPerm` or `DepPerm` (green bars) tend to request more permissions than those with code leakage (blue bars). For example, 13 add-ons request 10 permissions, and one

add-on requests as many as 13 permissions. These permissions can be exploited by attackers, compromising the security posture.

Case Study. One add-on, named *Sign*, with 26,841 users, was found to leak `DevPerm` and `DepPerm`. The permissions already authorized by user to this add-on are as follows:

1. Connect to an external service.
2. Read, edit, download, and permanently delete your contacts.
3. Display and run third-party web content in prompts and sidebars inside Google applications.
4. Read, edit, create and delete all your Google Docs documents.

With development and deployment privileges, attackers can modify the source code without the *genuine developer's* awareness and further deploy the modified malicious add-on into production environment. **From the add-on developer's perspective:** The attacker can redirect developer-owned external services, such as the subscription payment interface, to attacker-controlled services, enabling illicit profit. **From the add-on user's perspective:** The attacker can perform arbitrary actions using the modified add-on, such as downloading or deleting the victim user's contacts and documents.

5.3 Assessment of Leak of Secret

Normal workflow. As shown in Listing 1, in a secure and authorized implementation, the secret key used to access user-specific data on the service interface should be retrieved dynamically, requiring user authentication (lines 2–3). Even if the secret key is stored locally to avoid repeated logins, it must be securely stored using methods such as the workspace-provided API `PropertiesService.setProperty()`, which saves data in the workspace's cloud storage tied to the current user ID. This ensures that add-on front-end can only access the current user's secret key, preventing attackers from bypassing protections to retrieve confidential keys of other users.

```
1  // user login and authorize
2  confidential = login()
3  PropertiesService.setProperties("key", confidential);
4
5  function fetch_token(name) {
6      // extract the confidential from the save attribute
7      secret = PropertiesService.getProperity("key")
8      data = request(url, secret)
9      console.log(data);
10 }
```

Listing 1: Normal Access

Unauthorized Access and Attacks. Figure 4 illustrates the attack flow of secret key leakage and the resulting unauthorized access to service interfaces (upper right part). Based on the code leakage, an attacker can perform analysis on the leaked code to extract sensitive secret keys. These keys are used to authenticate and validate the connection to the service interface. This enables the attacker to exploit these secret keys to access or even manipulate critical service interfaces essential to the add-on developer. As shown in the leaked code in Listing 2, the developer hard-codes a secret key (`authorization` field in line 6) to access the service interface hosted by the add-on server. This practice enables unauthorized users to execute the URL request successfully (line 11) and retrieve - or even modify - sensitive information, as indicated by the POST request in line 3.

Notably, some APIs are explicitly listed in the add-on source code and expose the secret key. We also found other APIs, not mentioned in the code, using the same leaked key. For example, we examined the developer documentation at https://XXXX/api/documentation and identified additional API calls, like https://XXXX/api/v2/list, that also use the key (line 6 of Listing 2) for authorization. As prior studies on GitHub [41, 70] highlight, leaked service details (e.g., APIs or database schemas) broaden an attacker's surface, enabling further exploitation with the compromised key. This scenario allows tampering with add-on *DevOps* service interfaces and threatens other unrelated interfaces hosted by the same developer.

```
1  var options = {
2      'url': 'https://XXXX/api/v2/process',
3      'method': 'post',
4      'contentType': 'application/json',
5      'headers': {
6        'authorization': "Basic " + Utilities.base64Encode('mXXXX@gmail
             .com' + ":" + '544c606e286f4XXXXXac4b3359037f04'),
7      },
8      'muteHttpExceptions': true,
9      'payload': JSON.stringify(data)
10   };
11   var request = UrlFetchApp.fetch(options.url, options);
```

Listing 2: Secret Key Leakage

Dataset. Besides the code collected during the code leakage, we enhance our dataset by integrating code that developers may have released on public repository - GitHub. We employ the approach outlined in a previous study [39] to assist in crawling data from GitHub. The distinctive file structure of add-on repositories (e.g., the essential *appsscript.json* file) allows accurate identification of add-on projects on GitHub. We used the GitHub Search API [8] to retrieve all repositories containing this file. To overcome the API's 1,000-result limit, we partitioned the search by the size of *appsscript.json*, ensuring each group within the limit and retrieving all matching repositories. We adopt the established guidelines proposed by Munaiah et al. [44] to determine whether a repository qualifies as an engineered software project. After removing duplicates, we obtained a total of 1,608 open-source repositories of add-ons.

Challenges. Existing studies [29, 41] that detect secret key leakage primarily rely on already known patterns. As a result, they are limited to capturing secret keys for popular services, such as Amazon. In our case, this approach fails due to the diverse and developer-customized formats of secret keys. An initial approach is to identify all constant strings in the add-on code. However, our investigation reveals that not all constants are sensitive. For example, some constants represent publicly accessible resources, such as a privacy policy identified by a public ID (*1Hvb9QM0...-H-tF7*), which may resemble a secret key but pose no security risk. Therefore, identifying constant strings alone is insufficient. Understanding their context and dependencies is crucial to detecting actual secrets. To accurately detect secret key leakage, we must address two key challenges: **Challenge 1:** Identifying entry points for secret key usage: when the add-on connects to external services requiring authorization. **Challenge 2:** Minimizing false positives by reconstructing the secret key leakage path.

Undocumented web request API (C1). When the add-on projects access services provided by workspace, the only way is to use official APIs offered by workspace. However, when accessing the services

Table 4: Identified web request APIs

Sink	# Calls	Explanation	Hidden API
Confirmed *Yes*			
`UrlFetchApp.fetch()`	232	Recommended API to request web resource	✗
`OAuth2.createService()`	17	A customized library to authenticate the user	✓
`FirebaseApp.getDatabaseByUrl()`	7	A customized library to retrieve and store data on cloud	✓
`SpreadsheetApp.openBy*()`	18	Some developers use spreadsheet to store users' confidentials	✓
Confirmed *No*			
`CardService.*`	144	Create a card, but not the final destination of web request	*N.A.*
`HtmlService.createHtmlOutput()`	11	Create a HTML page, but not the final destination of web request	*N.A.*
`XmlService.getNamespace()`	8	Create a HTML page, but not the final destination of web request	*N.A.*

provided by add-on back-end, the approach becomes quite diverse. Although workspace provides the official API for add-on to connect and communicate with external service interfaces [17]. The add-on tends to utilize diverse approaches to access these service interfaces. Based on a manual verification of 50 add-ons, we find that some add-ons do not use the official API. They tend to utilize many custom libraries and other undocumented APIs (detailed soon in a later section) to support the connection. Such undocumented API usage must be addressed to avoid overlooking `AK` leaks and false negatives.

To accurately identify all undocumented web request APIs, we analyze the add-on code we extracted from code leakage, tracking all potential sinks to which HTTP requests flow (e.g., `HTTP://` or `HTTPS://`). We then sort these sinks by the count of appearances and compile a list of seven APIs as shown in Table 4. Through manual verification, we identify undocumented APIs that can be used to initiate web requests. The results are presented in Table 4. Ultimately, we identify three additional undocumented web request APIs beyond the recommended official API, `UrlFetchApp.fetch()`. The other three APIs, although they appear frequently, are not the final destination of the web request.

Secret key leakage path reconstruction (C2). We use backward taint analysis (from the taint sink) to capture leaked secret key `AK` when a web request occurs. We focus only on paths relevant to potential security risks, avoiding full input and output tracking, as well as implicit flow tracking typical of traditional information flow analysis. Specifically, we begin tracing from the point at which the web request is initiated, examining whether any secret key leakage paths can be reconstructed without relying on dynamic operation, such as user login operation. If we identify a path where all necessary parameters can be derived directly from the source code as demonstrated in Listing 2, we designate it as an secret key leakage path and output it as a candidate for potential leakage. To support this analysis, we use *CodeQL* [11], a static analysis tool developed by GitHub for JavaScript. We define the predefined sinks, as shown in Table 4, and trace back to potential sources. If we can trace back to a string constant, we identify it as a potential secret key leakage path.

Table 5: Examples of leaked secret keys

Name	Identified Leaked Value*
ApiKey	*398f9fac-1d64-4abc-8174-058b30XXXXXX*
AwsApiKey	*eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1fQ.XX-n0NBiXXXXXX*
ClientSecret	*b4_-B9fBF2zuuY650gXXXXXX*
Credential	*ENBqas6AmFlsYF9t67UEi-WFGTx_iYuaC_2MJ3PXXXXXX*
DevToken	*OzTAlzi33sZ1RedcmXXXXXX*
PrivateKey	*-BEGIN PRIVATE KEY–MIIG/QXXXXXX–END PRIVATE KEY-*
StripeToken	*sk_live_51KzqeqJKPo7p7uKAhbOzN3Ff00XXXXXX*
Secret	*Pqsuc5fJT52N0tDh387UVGZB9ry5xy9KegXXXXXX*
UserBaseKey	*1SeEDQsxOtmSySANYwOENGpdAkl2-cH0Qm7VhUXXXXXX*
UserPwd	*User@isheXXXXXX*

* We masked the actual values to prevent any potential harm to the developers.

Evaluation. Following the practice of existing studies [71], we randomly select 50 add-ons marked with leakage and 50 add-ons are not marked with leakage to verify the accuracy of our detection. No false negatives are found among the 50 add-ons marked with negative. However, we find two false positive secret keys in the marked add-ons. Our investigation reveals that the false positives were used to access publicly available websites, such as stock prices, rather than secret keys for sensitive service interfaces.

The evaluation result in Table 3 shows that secret key leakage is prevalent among currently vulnerable add-ons, including some with over 100,000 installations. In total, 96 add-ons, expose their secret keys in the add-on source code. Considering the attacker's capabilities are further expanded by the usage of secret key, it is critical for both developers and workspace to take action to strengthen the security of add-ons. We list some leaked secret keys in Table 5. These secret keys cover a broad range, including *ClientSecret*, *StripeToken*, and *PrivateKey*, none of which should be exposed to malicious attackers. The exposure is particularly concerning for leaked *StripeToken*, as its leakage could lead to financial loss for the add-on developer.

Additionally, we extract all external web requests from the vulnerable add-ons and identify the top 40 requested domain names, as shown in Table 7. These requests span a wide range of domain names, including frequently accessed ones like *google.com*, *github.com* and *googleapis.com*, which is expected as these services provide essential communication functions for add-ons. We also identify requests to more sensitive domain names related to finance (red text), education (blue text) and AI (purple text) services as shown in Table 7. Specifically, we observed requests to sensitive finance-related services like *stripe.com* 18 times, *moex.com* (a Russian financial website) 18 times, *paypal.com* 13 times. Requests were also sent to AI-related services, including 18 to *dreamstime.com* (an AI-powered image processing tool) and 17 to *kontent.ai* (an AI-powered headless content management system).

The most interesting findings arise when applying our tool to open-source GitHub repositories. Out of 1,608 add-ons analyzed, 613 add-ons could not be parsed by CodeQL. Among the 995 add-ons successfully parsed, only 40 show instances of secret key leakage. In contrast to the significant one-third leakage rate found among add-ons extracted from code leakage, this suggests that secret key leakage seems less prevalent in the GitHub dataset. However, manual inspection shows that the detected *leaked secrets* are all *masked placeholders*, not actual values, as shown in line 1 of Listing 3. None of the detected leaks contained real keys. By comparing secret key

Table 6: Partial records from the customer database retrieved using the leaked secret of the add-on project

Email	License Key	Date
andreas.XXXX@gmail.com	0F1ED4D4-XX-BE9D3553	05/05/2022
aloXXXX@cua.uam.mx	C111AD36-XX-2BEB2525	05/16/2022
XXXX@digitalmediaacademy.org	32265885-XX-EDFA8BD5	06/16/2022
jdaXXXX@twilio.com	7D70C174-XX-B589B806	11/14/2022
jonaXXXX@mosney.net	86B82A2C-XX-EB1232D8	02/23/2023
syarzXXXX@gmail.com	5EDA19A7-XX-55F6DFCC	04/22/2023
tranXXXX@gmail.com	37C50D4B-XX-E3FDD6C1	09/09/2023
drmcaXXXX@gmail.com	FC6CE274-XX-8B9F3D70	11/08/2023
XXXX@dariofinardi.it	99960837-XX-F4761662	12/27/2023
davidsamuXXXX@gmail.com	8CF6CBCD-XX-3DA91680	03/07/2024

We masked the *license key* and customer *email* to prevent any misuse of user data.

Table 7: Top 40 requested domain names

Domain Name	# Request	Domain Name	# Request
google.com	612	googleusercontent.com	22
googleapis.com	251	twitter.com	21
github.com	149	marketscreener.com	21
wikipedia.org	125	cloudfront.net	20
ecma-international.org	123	profit.co	20
mdn.io	63	instagram.com	19
stackoverflow.com	59	stripe.com	18
jsdelivr.net	51	serpapi.com	18
gstatic.com	48	moex.com	18
firebaseio.com	45	classcipe.com	18
momentjs.com	44	dreamstime.com	18
4-traders.com	42	smilebox.com	17
youtube.com	38	cloudfunctions.net	17
supermetrics.com	32	kontent.ai	17
talentsheets.com	30	microsoft.com	16
appspot.com	29	oracle.com	15
githubusercontent.com	28	codecogs.com	14
smartdraw.com	27	nearpod.com	14
apache.org	23	lodash.com	14
techtracker.io	23	paypal.com	13

red text: finance domain blue text: education domain purple text: AI domain

leaks in production add-ons and on GitHub, we find GitHub developers rarely expose keys, likely due to awareness of *public* repositories [29, 36, 37, 41]. In contrast, add-on developers may assume workspace-managed code is private. Prior studies [23, 27, 54, 55, 57] show developers trust vendors to securely handle code.

```
const PRIVATE_KEY = '## YOUR PRIVATE KEY ##';
// e.g. '-----BEGIN PRIVATE KEY-----\NCuY2TN1M1c...0bKG/NvOt\n
      -----END PRIVATE KEY-----\n';
```

Listing 3: Example of masked secret key placeholder

Case Study. An add-on named *Codes*, with 810 users, last modified on April 1, 2024, is detected with secret key leakage. Specifically, the developer encoded their secret key (e.g., *1X8QOlbWWIFWzXXXX*) directly in the add-on code to access the customer database. By reconstructing the secret key leakage path, we were able to retrieve the full customer database. A small portion of the database information is shown in Table 6, with confidential details masked for ethical considerations. The users of this add-on span a wide range, not limited to Gmail users. For example, users from domains such as *cua.uam.mx*, *digitalmediaacademy.org*, *twilio.com*, *mosney.net*, and *dariofinardi.it* have also exposed their confidential information to malicious attackers.

Even worse, this secret key grants attackers the ability to add new customers, modify or delete existing ones. In other words, attackers gain admin privileges over the developer's database through the leaked secret key. The attacker can compromise these valuable customers by simply removing them from the database. If the developer does not maintain a backup or execution history log, the lost data cannot be recovered.

5.4 Trend of Exposure and Developer Response

Our evaluation dataset was initially crawled during the code leakage phase at the beginning of 2024. To reveal the time trend of these exposures, we revisited these add-ons at the end of 2024. Surprisingly, we found that around 23 add-ons are no longer accessible. Of these, 9 add-ons now prompt an access permission request, displaying an *Access Denied* message, while 14 add-ons show an error message, such as *Error 404 (Not Found)* or *Error 500 (Server Error)*. This re-visiting suggests that some developers have become aware of potential vulnerabilities and have restricted access permissions to prevent malicious access. However, most developers remain unaware of the potential leakage and vulnerabilities present in their add-ons. We then contacted affected developers to disclose the vulnerabilities, and many confirmed our findings and expressed concerns about potential abuse. A sample response is shown below: "*Hmmm, that's pretty bad. I didn't realize that was accessible to anyone. Please don't abuse it or share it—I will try to patch it soon!*"

6 Discussions

6.1 Limitations

Although we perform automated analysis and detection for vulnerabilities in the DevOps of add-on projects, our work still has some limitations. First, our tool utilizes the existing CodeQL parser for static code analysis. Like all static analysis tools designed for standard code, it is unable to parse code that is heavily obfuscated. Second, our secret key leakage analysis is based on the code leakage of the add-on project. Thus, we are unable to analyze all add-ons for which we cannot fetch the source code. Third, we construct the hidden APIs based on an empirical study, and we apply manual efforts to identify the potential `APIs` used for web service requests. In the future, more diverse `APIs` may be used by add-on developers and should be added to the library, but this will require only a one-time effort. Finally, we focus primarily on detecting leaks from the developers' perspective (*DevOps*) without addressing all possible attacks that could occur within this ecosystem.

6.2 Lessons Learned and Mitigation

Vetting of add-on. Workspace enforces a strict vetting process [12]. However, vendors do not disclose their vetting criteria for researchers to evaluate. Although the vendor provides documentation [12] for add-on developers, these documents do not discuss the potential issues that developers may face when considering the DevOps of add-on project. The workspace exposes confidential IDs to end-users and lacks proper developer verification, resulting in both code and permission leaks. Thus, even if add-on developers strictly follow the guidelines, their projects remain vulnerable. While developers are responsible for the secret key leakage, it remains unclear how such risky practices pass the vetting process. One possible explanation is that both developers and workspace *believe* the add-on projects are protected and cannot be leaked to attackers.

For workspace developer.

Implement stricter permission management: A GitHub-similar request and approval mechanism can safeguard the development and deployment of the add-on project. For instance, whenever a collaborator wants to merge their modifications into the active version, they should request approval from the project admin, with

a similar process for deployment. This stricter control over updates and deployment allows benign developers to quickly identify and respond to any malicious activity before it impacts the production environment. Alternatively, the production environment should be only accessible and managed by the admin.

Support secure authentication: For web service requests, workspace provides only a basic API, `UrlFetchApp.fetch()`, which cannot ensure secure network requests. Many add-on developers just hard-code their secret keys into URL parameters due to the lack of authentication mechanism supported by workspace. A robust authentication mechanism for web requests [65, 70], such as OAuth [33] with dynamic user login, can be implemented. The workspace can utilize encryption and distribute the secret only to add-on developers for encrypting data transfers.

Review and inspect the add-on project: The workspace can run static analysis or security checks before deployment to detect potential leaks. Techniques like suspicious API usage scanning [57] and secret key tracing [69] can be adapted to detect sensitive data flows and alert developers before production deployment.

For add-on developer.

Secure service interface interaction: Developers should never embed secret keys in source code. Instead, store secrets in environment variables or use tools such as AWS Secrets Manager [20] and Google Secret Manager [31]. For server requests, use standard authentication methods such as OAuth 2.0 or JWT [22] to ensure secure access. Additionally, obfuscation tools such as UglifyJS [52] and ProGuard [32] help protect add-ons.

Identify the source of requests: Developers can verify request origins by adding add-on-specific metadata, such as a custom header (e.g., `X-App-ID`) or an HMAC token [43, 51]. Servers validate these to distinguish legitimate from unauthorized requests, with added protections like rate limiting or timestamp checks [48].

For software engineering researchers. Our work provides a developer-centered security assessment of the workspace add-on DevOps lifecycle, complementing prior user-focused studies [34, 42, 61, 68, 70], and highlights the need to strengthen cloud-based collaborative DevOps models.

Design secure architectures and protocols: Researchers could design secure architectures that integrate fine-grained, role-based access control with context-aware policy enforcement [26], dynamically restricting add-on permissions according to the developer's role and task. Our findings also suggest new directions for protocol design, such as incorporating mutual authentication [70] between developers and workspace services, and defining explicit permission negotiation protocols to prevent unintended privilege escalations.

Real-time monitoring: Real-time monitoring modules should be architected to inspect CI/CD pipelines for anomalous behaviours [36], including unauthorised API calls or suspicious privilege changes [24].

7 Related Work

DevOps security. DevOps security is a critical focus of software engineering. Prior studies have examined DevOps security in Android [24, 57], ChatGPT [65], and GitHub CI/CD [29, 72] applications. We provide a comprehensive comparison with existing work in Table 8. They are all vulnerable to secret key leakage, enabling unauthorized access. Furthermore, ChatGPT GPTs leak code, while GitHub Actions leak permissions.

Table 8: Comparison of DevOps security studies

Platform	Application	Leakage Type		
		Code	Permission	Secret Key
Android	Mini-programs			✓
ChatGPT	GPTs	✓		✓
GitHub CI/CD	Actions		✓	✓
Workspace	**Add-ons**	✓	✓	✓

Web App Security. Web security has always been a prominent topic, particularly with the rise of team collaboration features [53, 56]. Wang et al. [61] systematically revealed unauthorized origin crossing during communication between web services and mobile apps, a design flaw that can result in severe impacts, including hijacking attacks. Hazard [68] analyzed the ecosystem of popular Team Chat Systems like Slack and discovered vulnerabilities such as invocation hijacking, message monitoring, and improper permission isolation. Wan et al. [54] revealed that many vulnerabilities stem from web resource sharing.

Mini-program Security. The security of mini-programs installed on Android Super Apps like WeChat [3], Alipay [2], and TikTok [9] have garnered significant attention in the research community. Although mini-programs run within Android Super Apps, they resemble add-ons in the workspace. Zhang et al. [69] were the first to crawl and analyze the features of WeChat mini-programs on a large scale. Wang et al. [58, 60] discovered permission inconsistencies between the mobile and desktop versions of Super Apps. TaintMini[57] used static analysis to build mini-program data flow graphs, showing that cross-mini-program data flows may leak sensitive data without authentication. WeMinT [42] and AppSecret [70] both highlighted the widespread issue of secret key leakage in mini-programs. MiniCAT [71] explored the routing mechanisms and sharing features of mini-programs, uncovering vulnerabilities to *Cross-Page Request Forgery* attacks with severe consequences. APIScope [59] found that many mini-programs use undocumented APIs to bypass Android permissions, leading to privilege escalation.

8 Conclusion

The add-on built on workspace introduces new attack surfaces for both web applications and end-users. We are the first to systematically uncover the DevOps models of add-ons. Our study demonstrates that improper permission management designed by the workspace can lead to severe impacts when exploited by attackers. To assess the prevalence of these vulnerabilities, we developed a detector that analyzed 5,300 add-ons. Our results show that 274 add-ons are vulnerable to code leakage, while 96 have secret key leakage. We illustrate the potential consequences of these weaknesses through real-world attack case studies, and propose mitigations for both workspace providers and add-on developers. We hope our work helps enhance the ecosystem's security.

Availability. The source code is available online [16]. Given the dataset's sensitivity, and following requests from our ethics committee and affected developers, we have not released it publicly.

Acknowledgments

We would like to thank anonymous reviewers for improving this manuscript. This research has been partially supported by the Australian Research Council Discovery Projects (DP240103068).

References

[1] 2022. Microsoft Threat Modeling Tool threats. https://learn.microsoft.com/en-us/azure/security/develop/threat-modeling-tool-threats. Online; Accessed: August 25, 2022.

[2] 2024. AliPay Mini-Programs. https://global.alipay.com/platform/site/product/mini-program. Online; Accessed: March 10, 2025.

[3] 2024. Connecting a billion people with calls, chats, and more. https://www.wechat.com/AU. Online; Accessed: March 10, 2025.

[4] 2024. Create, connect and collaborate with the power of AI. https://workspace.google.com/intl/en_au/. Online; Accessed: March 10, 2025.

[5] 2024. GitHub for Google Chat. https://workspace.google.com/marketplace/app/github_for_google_chat/536184076190. Online; Accessed: March 10, 2025.

[6] 2024. GPT for Gmail. https://workspace.google.com/marketplace/app/gpt_for_gmail/802100925247. Online; Accessed: March 10, 2025.

[7] 2024. Market Share of OneDrive and G Suite. https://6sense.com/tech/document-management/onedrive-market-share. Online; Accessed: March 10, 2025.

[8] 2024. Requesting more than 1000 items from Github Search. https://github.com/orgs/community/discussions/64629. Online; Accessed: March 10, 2025.

[9] 2024. TikTok Mini-Programs. https://www.tiktok.com/discover/mini-programs. Online; Accessed: March 10, 2025.

[10] 2024. Zapier for Google Chat. https://workspace.google.com/marketplace/app/zapier_for_google_chat/905224480607. Online; Accessed: March 10, 2025.

[11] 2025. About code scanning with CodeQL. https://docs.github.com/en/code-security/code-scanning/introduction-to-code-scanning/about-code-scanning-with-codeql. Online; Accessed: March 10, 2025.

[12] 2025. Build Google Workspace Add-ons. https://developers.google.com/apps-script/add-ons/how-tos/building-workspace-addons. Online; Accessed: March 10, 2025.

[13] 2025. Create a Google Cloud project. https://developers.google.com/workspace/guides/create-project. Online; Accessed: March 10, 2025.

[14] 2025. Market Share of Slack. https://6sense.com/tech/productivity/slack-market-share. Online; Accessed: March 10, 2025.

[15] 2025. Microsoft OneDrive. https://www.microsoft.com/en-au/microsoft-365/onedrive/online-cloud-storage. Online; Accessed: March 10, 2025.

[16] 2025. Understanding DevOps Security of Google Workspace Apps (Source Code). https://anonymous.4open.science/r/DevOps-D82B/. Online; Accessed: March 10, 2025.

[17] 2025. UrlFetchApp — Apps Script: Fetch resources and communicate with other hosts over the Internet. https://developers.google.com/apps-script/reference/url-fetch/url-fetch-app. Accessed: March 10, 2025.

[18] 2025. Welcome to Google's Bug Hunting community. https://bughunters.google.com. Online; Accessed: March 10, 2025.

[19] Yasemin Acar, Christian Stransky, Dominik Wermke, Charles Weir, Michelle L Mazurek, and Sascha Fahl. 2017. Developers need support, too: A survey of security advice for software developers. In *2017 IEEE Cybersecurity Development (SecDev)*. IEEE, 22–26.

[20] Amazon Web Services. 2025. AWS Secrets Manager. https://aws.amazon.com/secrets-manager/. Online; Accessed: July 01, 2025.

[21] Anthony Andreoli, Anis Lounis, Mourad Debbabi, and Aiman Hanna. 2023. On the prevalence of software supply chain attacks: empirical study and investigative framework. *Forensic Science International: Digital Investigation* 44 (2023), 301508.

[22] Auth0. 2025. JWT.IO - JSON Web Tokens. https://jwt.io. Online; Accessed: July 01, 2025.

[23] David G Balash, Xiaoyuan Wu, Miles Grant, Irwin Reyes, and Adam J Aviv. 2022. Security and privacy perceptions of Third-Party application access for google accounts. In *31st USENIX security symposium (USENIX Security 22)*. 3397–3414.

[24] Islem Bouzenia and Michael Pradel. 2024. Resource usage and optimization opportunities in workflows of github actions. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 1–12.

[25] Thanh Bui, Siddharth Rao, Markku Antikainen, and Tuomas Aura. 2020. Xss vulnerabilities in cloud-application add-ons. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*. 610–621.

[26] Supriyo Chakraborty, Chenguang Shen, Kasturi Rangan Raghavan, Yasser Shoukry, Matt Millar, and Mani Srivastava. 2014. ipShield: A Framework For Enforcing Context-Aware Privacy. In *11th USENIX symposium on networked systems design and implementation (NSDI 14)*. 143–156.

[27] Yunang Chen, Yue Gao, Nick Ceccio, Rahul Chatterjee, Kassem Fawaz, and Earlence Fernandes. 2022. Experimental security analysis of the app model in business collaboration platforms. In *31st USENIX Security Symposium (USENIX Security 22)*. 2011–2028.

[28] Sascha Fahl, Marian Harbach, Thomas Muders, Lars Baumgärtner, Bernd Freisleben, and Matthew Smith. 2012. Why Eve and Mallory love Android: An analysis of Android SSL (in) security. In *Proceedings of the 2012 ACM conference on Computer and communications security*. 50–61.

[29] Runhan Feng, Ziyang Yan, Shiyan Peng, and Yuanyuan Zhang. 2022. Automated detection of password leakage from public github repositories. In *Proceedings of the 44th International Conference on Software Engineering*. 175–186.

[30] Google. 2025. Google Apps Script Samples. https://github.com/googleworkspace/apps-script-samples. Online; Accessed: March 10, 2025.

[31] Google Cloud. 2025. Google Secret Manager. https://cloud.google.com/secret-manager. Online; Accessed: July 01, 2025.

[32] Guardsquare. 2025. ProGuard: Java and Android code shrinker, optimizer, obfuscator, and pre-verifier. https://www.guardsquare.com/products/proguard. Online; Accessed: July 01, 2025.

[33] D. Hardt. 2012. The OAuth 2.0 Authorization Framework. RFC 6749. Online; Accessed: July 01, 2025.

[34] Hamza Harkous and Karl Aberer. 2017. " If You Can't Beat them, Join them" A Usability Approach to Interdependent Privacy in Cloud Apps. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*. 127–138.

[35] Jason Huggins. 2025. Selenium website. https://www.selenium.dev/. Online; Accessed: March 10, 2025.

[36] Igibek Koishybayev, Aleksandr Nahapetyan, Raima Zachariah, Siddharth Muralee, Bradley Reaves, Alexandros Kapravelos, and Aravind Machiry. 2022. Characterizing the security of github CI workflows. In *31st USENIX Security Symposium (USENIX Security 22)*. 2747–2763.

[37] Alexander Krause, Jan H Klemmer, Nicolas Huaman, Dominik Wermke, Yasemin Acar, and Sascha Fahl. 2023. Pushed by Accident: A Mixed-Methods Study on Strategies of Handling Secret Information in Source Code Repositories. In *32nd USENIX Security Symposium (USENIX Security 23)*. 2527–2544.

[38] Shuai Li, Zhemin Yang, Nan Hua, Peng Liu, Xiaohan Zhang, Guangliang Yang, and Min Yang. 2022. Collect responsibly but deliver arbitrarily? a study on cross-user privacy leakage in mobile apps. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 1887–1900.

[39] Song Liao, Long Cheng, Haipeng Cai, Linke Guo, and Hongxin Hu. 2023. SkillScanner: Detecting Policy-Violating Voice Applications Through Static Analysis at the Development Phase. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. 2321–2335.

[40] Yuxi Ling, Kailong Wang, Guangdong Bai, Haoyu Wang, and Jin Song Dong. 2022. Are they toeing the line? diagnosing privacy compliance violations among browser extensions. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–12.

[41] Michael Meli, Matthew R McNiece, and Bradley Reaves. 2019. How bad can it git? characterizing secret leakage in public github repositories.. In *NDSS*.

[42] Shi Meng, Liu Wang, Shenao Wang, Kailong Wang, Xusheng Xiao, Guangdong Bai, and Haoyu Wang. 2023. WeMinT: Tainting Sensitive Data Leaks in WeChat Mini-Programs. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1403–1415.

[43] Vladislav Mladenov, Christian Mainka, and Jörg Schwenk. 2015. On the security of modern single sign-on protocols: Second-order vulnerabilities in openid connect. *arXiv preprint arXiv:1508.04324* (2015).

[44] Nuthan Munaiah, Steven Kroh, Craig Cabrey, and Meiyappan Nagappan. 2017. Curating github for engineered software projects. *Empirical Software Engineering* 22 (2017), 3219–3253.

[45] Shradha Neupane, Grant Holmes, Elizabeth Wyss, Drew Davidson, and Lorenzo De Carli. 2023. Beyond typosquatting: an in-depth look at package confusion. In *32nd USENIX Security Symposium (USENIX Security 23)*. 3439–3456.

[46] npm, Inc. 2018. Details about the event-stream incident. https://blog.npmjs.org/post/180565383195/details-about-the-event-stream-incident. Online; Accessed: July 01, 2025.

[47] Marc Ohm, Henrik Plate, Arnold Sykosch, and Michael Meier. 2020. Backstabber's knife collection: A review of open source software supply chain attacks. In *Detection of Intrusions and Malware, and Vulnerability Assessment: 17th International Conference, DIMVA 2020, Lisbon, Portugal, June 24–26, 2020, Proceedings 17*. Springer, 23–43.

[48] OWASP Foundation. 2023. OWASP API Security Top 10 - 2023 Edition. https://owasp.org/API-Security/editions/2023/en/0x00-header/. Accessed: 2025-07-14.

[49] Ita Ryan, Utz Roedig, and Klaas-Jan Stol. 2023. Unhelpful assumptions in software security research. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. 3460–3474.

[50] Snyk Security Team. 2018. A post-mortem of the malicious event-stream backdoor. https://snyk.io/blog/a-post-mortem-of-the-malicious-event-stream-backdoor/. Online; Accessed: July 01, 2025.

[51] James M Turner. 2008. The keyed-hash message authentication code (hmac). *Federal Information Processing Standards Publication* 198, 1 (2008), 1–13.

[52] UglifyJS Contributors. 2025. UglifyJS: JavaScript parser, mangler/compressor and beautifier toolkit. https://github.com/mishoo/UglifyJS. Online; Accessed: July 01, 2025.

[53] Liuhuo Wan, Kailong Wang, Kulani Mahadewa, Haoyu Wang, and Guangdong Bai. 2024. Don't Bite Off More than You Can Chew: Investigating Excessive Permission Requests in Trigger-Action Integrations. In *Proceedings of the ACM Web Conference 2024*. 3106–3116.

[54] Liuhuo Wan, Kailong Wang, Haoyu Wang, and Guangdong Bai. 2024. Is It Safe to Share Your Files? An Empirical Security Analysis of Google Workspace. In *Proceedings of the ACM on Web Conference 2024*. 1892–1901.

[55] Liuhuo Wan, Chuan Yan, Mark Huasong Meng, Kailong Wang, and Haoyu Wang. 2024. Analyzing Excessive Permission Requests in Google Workspace Add-Ons. In *International Conference on Engineering of Complex Computer Systems*. Springer, 323–345.

[56] Liuhuo Wan, Yanjun Zhang, Ruiqing Li, Ryan Ko, Louw Hoffman, and Guangdong Bai. 2022. SATB: A Testbed of IoT-Based Smart Agriculture Network for Dataset Generation. In *International Conference on Advanced Data Mining and Applications*. Springer, 131–145.

[57] Chao Wang, Ronny Ko, Yue Zhang, Yuqing Yang, and Zhiqiang Lin. 2023. Taintmini: Detecting flow of sensitive data in mini-programs with static taint analysis. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 932–944.

[58] Chao Wang, Yue Zhang, and Zhiqiang Lin. 2023. One Size Does Not Fit All: Uncovering and Exploiting Cross Platform Discrepant APIs in WeChat. In *32nd USENIX Security Symposium (USENIX Security 23)*. 6629–6646.

[59] Chao Wang, Yue Zhang, and Zhiqiang Lin. 2023. Uncovering and exploiting hidden apis in mobile super apps. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. 2471–2485.

[60] Chao Wang, Yue Zhang, and Zhiqiang Lin. 2024. RootFree Attacks: Exploiting Mobile Platform's Super Apps From Desktop. In *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security*. 830–842.

[61] Rui Wang, Luyi Xing, XiaoFeng Wang, and Shuo Chen. 2013. Unauthorized origin crossing on mobile platforms: Threats and mitigation. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. 635–646.

[62] Zihan Wang, Zhongkui Ma, Xinguo Feng, Ruoxi Sun, Hu Wang, Minhui Xue, and Guangdong Bai. 2024. Corelocker: Neuron-level usage control. In *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2497–2514.

[63] Chuan Yan, Bowei Guan, Yazhi Li, Mark Huasong Meng, Liuhuo Wan, and Guangdong Bai. 2025. Understanding and Detecting File Knowledge Leakage in GPT App Ecosystem. In *Proceedings of the ACM on Web Conference 2025*. 3831–3839.

[64] Chuan Yan, Mark Huasong Meng, Fuman Xie, and Guangdong Bai. 2024. Investigating documented privacy changes in android os. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 2701–2724.

[65] Chuan Yan, Ruomai Ren, Mark Huasong Meng, Liuhuo Wan, Tian Yang Ooi, and Guangdong Bai. 2024. Exploring chatgpt app ecosystem: Distribution, deployment and security. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. 1370–1382.

[66] Chuan Yan, Liuhuo Wan, Bowei Guan, Fengqi Yu, and Guangdong Bai. 2025. Tracking gpts third party service: Automation, analysis, and insights. In *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering*. 1602–1606.

[67] Yuqing Yang, Yue Zhang, and Zhiqiang Lin. 2025. Understanding Miniapp Malware: Identification, Dissection, and Characterization. In *Proceedings 2025 Network and Distributed System Security Symposium. San Diego, CA, USA*.

[68] Mingming Zha, Jice Wang, Yuhong Nan, Xiaofeng Wang, Yuqing Zhang, and Zelin Yang. 2022. Hazard Integrated: Understanding Security Risks in App Extensions to Team Chat Systems.. In *NDSS*.

[69] Yue Zhang, Bayan Turkistani, Allen Yuqing Yang, Chaoshun Zuo, and Zhiqiang Lin. 2021. A measurement study of wechat mini-apps. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 5, 2 (2021), 1–25.

[70] Yue Zhang, Yuqing Yang, and Zhiqiang Lin. 2023. Don't leak your keys: Understanding, measuring, and exploiting the appsecret leaks in mini-programs. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. 2411–2425.

[71] Zidong Zhang, Qinsheng Hou, Lingyun Ying, Wenrui Diao, Yacong Gu, Rui Li, Shanqing Guo, and Haixin Duan. 2024. MiniCAT: Understanding and Detecting Cross-Page Request Forgery Vulnerabilities in Mini-Programs. In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security*.

[72] Lida Zhao, Sen Chen, Zhengzi Xu, Chengwei Liu, Lyuye Zhang, Jiahui Wu, Jun Sun, and Yang Liu. 2023. Software composition analysis for vulnerability detection: An empirical study on Java projects. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 960–972.

[73] Yajin Zhou and Xuxian Jiang. 2012. Dissecting android malware: Characterization and evolution. In *2012 IEEE symposium on security and privacy*. IEEE, 95–109.

[74] Markus Zimmermann, Cristian-Alexandru Staicu, Cam Tenny, and Michael Pradel. 2019. Small world with high risks: A study of security threats in the npm ecosystem. In *28th USENIX security symposium (USENIX Security 19)*. 995–1010.