Beyond the Scope: Security Testing of Permission Management in Team Workspace

Liuhuo Wan, Chuan Yan, Mark Huasong Meng, Kailong Wang, Haoyu Wang, Guangdong Bai, Jin Song Dong

***Abstract*—Nowadays team workspaces are widely adopted for multi-user collaboration and digital resource management. To further broaden real-world applications, mainstream team workspaces platforms, such as Google Workspace and Microsoft OneDrive, allow third-party applications (referred to as *add-ons*) to be integrated into their workspaces, significantly extending the functionality of team workspaces. The powerful multi-user collaboration capabilities and integration of *add-ons* make team workspaces a central hub for managing shared resources and protecting them against unauthorized access. Due to the collaboration features of team workspaces, add-ons involved in collaborations may bypass the permission isolation enforced by the administrator, unlike in single-user permission management.**

This paper aims to investigate the permission management landscape of team workspaces add-ons. To this end, we perform an in-depth analysis of the enforced access control mechanism inherent in this ecosystem, considering both multi-user and cross-app features. We identify three potential security risks that can be exploited to cause permission escalation. We then systematically reveal the landscape of permission escalation risks in the current ecosystem. Specifically, we propose an automated tool, *TAI*, to systematically test all possible interactions within this ecosystem. Our evaluation reveals that permission escalation vulnerabilities are widespread in this ecosystem, with 41 interactions identified as problematic. Our findings should raise an alert to both the team workspaces platforms and third-party developers.

I. Introduction

Team workspaces offer a comprehensive toolkit to streamline business operations. For example, users can manage the company's daily tasks through a spreadsheet. Other users can actively access this spreadsheet, assuming various roles such as viewer, commenter, or editor. Prominent examples of team workspaces include Google Workspace [1] and Microsoft OneDrive [2]. The demand for online and remote collaborations, particularly exacerbated by the effects of the pandemic, has propelled the widespread adoption of these team workspaces. Google Workspace, for example, boasts over three billion users [3] and is adopted by nine million companies as their business solution. Team workspaces can be further enhanced by integrating third-party applications (named add-ons) to supplement advanced features. These add-ons integrate with external services such as Confluence, Evernote, GPT, Zoom, Dropbox, Webex, etc. While such integration enhances the user experience, it also introduces complexities in permission management when accessing sensitive resources hosted by team workspaces. The lax management of add-on access control inevitably introduces new attack surfaces, posing risks to security-critical collaborations. Despite its significance, this issue has not been thoroughly studied before.

A add-on accesses resources managed by team workspaces through host APIs, for tasks such as viewing documentation or replying to an email. These host APIs are regulated by different permissions. The add-on employs a two-level permission management system, comprising OAuth permission scope checking and user role checking. In detail, during installation, the add-on requests OAuth tokens with permission scope, which are used to interact with resources managed by team workspaces. For example, an add-on might request the OAuth permission scope to '*view and edit all Google Documents*' from the user. However, some resources accessed by the add-on during runtime may not be owned by the user. If the user is assigned the viewer role and cannot edit the resource, the add-on installed by that user is similarly prohibited from modifying the resource.

Shadow of security risks. Team workspaces supports multi-user collaboration through role-based permission management, where an owner retains full privileges and assigns partial privileges to other roles as shown in Figure 1. It is expected that a user role does not exceed the permission scopes assigned by the owner. Similarly, add-ons installed by specific a user role (e.g., viewer), should adhere to the permission scopes delegated by this user. This raises a critical question: does permission management remain consistent across different users and add-ons. For example, *can an unprivileged member install an add-on to access resources they are not entitled to, such as hidden rows or columns in a protected Spreadsheet, and even modify them?* It is important to note that the installation of add-ons in the private workspace (e.g., personal workspace) typically requires no vetting process [4]. In such scenarios, if permission management is not well-established, users could potentially achieve permission escalation with the assistance of add-ons.

Despite the potential security risks brought about by add-ons, limited efforts [4]–[6] have been devoted to understanding the security issues they pose due to the black-box nature of both team workspaces and add-ons. Furthermore, the complexity of team workspaces's features, including multi-user roles, complicates the analysis and testing of these security risks. Existing efforts [4]–[6] predominantly rely on manual interactions and inspections. Consequently, **there is a need**

- *L. Wan is with The University of Queensland, Australia.*
- *C. Yan is with The University of Queensland, Australia.*
- *M. H. Meng is with The Technical University of Munich, Germany.*
- *K. Wang is with Huazhong University of Science and Technology, China.*
- *H. Wang is with Huazhong University of Science and Technology, China.*
- *G. Bai is with The University of Queensland, Australia and also with National University of Singapore, Singapore.*
- *J. S. Dong is with National University of Singapore.*

for automated tools to test and analyze permission management considering the integration of add-ons.

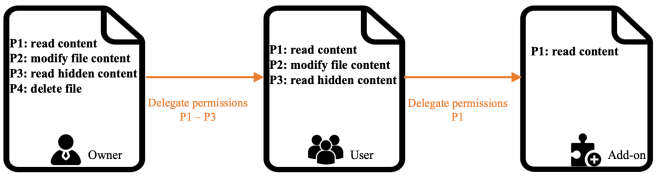

Fig. 1: Permission delegation demonstration

To comprehensively evaluate the underlying permission escalation problem caused by the add-ons, We systematically test and analyze the add-ons' capability to utilize host APIs for achieving permission escalation. Specifically, we develop a tool named *TAI* to automatically generate test cases and evaluate the add-ons' capability under different permission scopes. *TAI* identifies 41 risky host APIs that add-on can exploit, leading to permission escalation. These APIs affect prominent services like Spreadsheets, Documents, Drive Files, and Forms. They allow unauthorized operations on resources, including viewing hidden data protected by the admin, unauthorized modifications of specific data groups, and interfering with existing members of shared resources.

Contributions. The contributions of this work are summarized as follows:

- **An in-depth understanding of the permission system.** We conduct a detailed analysis and reveal the complex permission control mechanisms within team workspaces: the two-level permission checking for add-ons. This enables us to understand the precise mechanisms that govern data access and permission requests. Based on this established knowledge, we can identify and formally define the permission escalation vulnerabilities in the current system.
- **A systematic tool for the permission escalation testing.** To comprehensively detect problematic add-ons' usage of host APIs that could lead to permission escalation, we have devised and implemented an automated tool called *TAI*. *TAI* is built on a pipeline that includes permission categorization, test case generation, and user-role-based testing. Using *TAI*, we can detect discrepancies during the permission-checking process. This systematic scanning approach enables us to identify all risky API usages within the ecosystem.
- **The landscape of permission escalation issues.** Our research has uncovered a large amount of problematic APIs. Through detailed discussions supported by concrete case studies, we have identified potential attack scenarios that could arise from problematic API usage. Furthermore, we have proposed countermeasures to mitigate the permission escalation issues for team workspaces and end-users.

Ethics and Disclosure. All our experiments are conducted using test accounts. The workspace is controlled, with the authors as the only members. The malicious add-ons designed are only installed in the controlled team workspaces and access limited resources. We did not distribute these malicious add-ons into other team workspaces or the public marketplace. All our attacks would not affect users and resources other than the authors' testing accounts. We ethically disclosed our findings to team workspaces and received acknowledgment.

II. PROBLEM FORMULATION

A. Background and the Permission system

Before the launch of team workspaces, collaboration in organizations often relied on a combination of tools that, were not as seamlessly integrated or cloud-based as modern solutions. Different applications often require manual syncing and are not always designed for real-time, cloud-based collaboration. For example, Dropbox for file storage or Slack for messaging, operate in separate environments, requiring users to juggle multiple applications and interfaces. This fragmentation make it challenging to maintain a cohesive workflow, as data has to be manually transferred, and users have to manage different accounts and logins. Team workspaces bring a more integrated, cloud-native approach by combining email, chat, file storage, document editing, and scheduling tools into a single, unified platform. This integration allows for real-time collaboration on documents, seamless communication via email, and centralized file storage and sharing—all accessible through a web browser without downloading separate desktop applications. As a result, team workspaces significantly simplify workflows, reduce the reliance on disparate tools and create a more streamlined, efficient collaboration environment for many organizations.

Sharing is one of the most powerful features supported by team workspaces. Users can seamlessly share their resources and engage in real-time collaborative file editing, eliminating the need for redundant resource distribution.

Users. There are four types of collaborators supported in the current ecosystem of team workspaces, owner, editor, commenter and viewer. The owner has all privileges related to shared resources, while other roles have varying permission scopes, as suggested by their names. The owner can assign different roles to the collaborators.

Add-ons. The users can integrate add-ons into their team workspaces. Beyond interactions with legitimate users, add-ons can access resources through user delegation as shown in Figure 1. With permission scopes granted by users, add-ons can access and manipulate resources stored within the user's workspace. These resources fall into two categories: those for which this user is the owner and those for which this user is a collaborator (e.g., viewer or editor).

B. Permission System

Objects. There is a list of resources supported by the team workspaces, such as documents, spreadsheets, presentations, and forms. Each resource exhibits a hierarchical structure composed of finer-grained components. For example, a spreadsheet consists of rows, each of which contains multiple cells. Every cell may carry its own set of attributes, such as content value, background color, and formatting. We refer to all of these components, at any level of granularity, as objects.

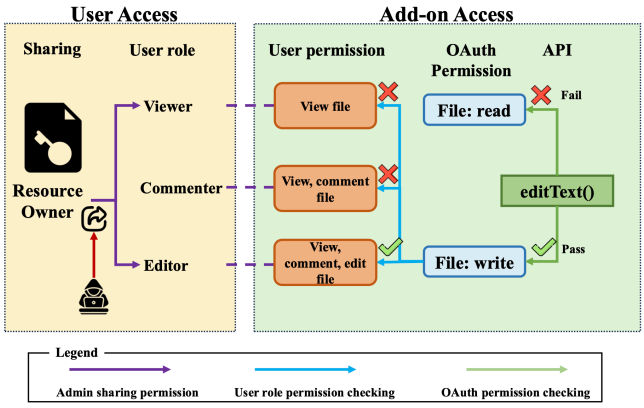

Fig. 2: Access control mechanisms in the current ecosystem

Subjects. In total, team workspaces defines three types of subjects: (1) the **owner** of the object, (2) **collaborators** on the object (may be viewer, commenter or editor), and (3) **add-ons** that can access predefined objects.

Permissions. We represent each permission in team workspaces using a tuple $(subject, operation, object)$, which indicates who (the subject) can perform what action (the operation) on which resource (the object). The team workspaces support five types of operation: *create, view, comment, modify*, and *delete*. These permissions may be applied at varying levels of granularity, ranging from an entire `Document` to nested elements such as a `Footnote` within that `Document`. For example, the permissions $(add\text{-}on, delete, \texttt{Document})$ and $(add\text{-}on, delete, \texttt{Footnote})$ indicate the capability of an add-on to delete the entire `Document` or only the `Footnote`, respectively.

Team workspaces enforce a two-level access control model for add-ons, as illustrated in Figure 2. When the add-on utilizes host APIs like `editText()` to access the installer's resources, team workspaces verify the authorized ***OAuth*** permission scopes granted by the installer and the installer's ***role*** (collaborator's permissions) associated with the resource.

*a) **Level 1: OAuth permission scope checking**:* As illustrated by the green lines in Figure 2, when the add-on tries to execute the API `editText()`, team workspaces first check the already authorized OAuth permission scopes by its installer. If the add-on has been granted `file:write` permission, it is allowed to write to the file and successfully passes the first-level permission checking. In contrast, if the add-on is only authorized with `file:read` permission, it will fail to execute the `editText()` API due to insufficient permissions.

*b) **Level 2: User role permission checking**:* After successfully completing the first-level permission check, team workspaces then evaluate the installer's role on the resource. As shown by the blue lines in Figure 2, for the `editText()` API, even if the installer has granted the add-on with `file:write` permission, the second-level permission check will fail if the installer's role for the resource is set to *viewer* or *commenter* without editing privileges. As a result, team workspaces returns an exception.

Based on this two-level permission checking, we can conclude that the add-on is constrained by both granted permission scope and installers' roles: $\mathbb{P}_{\text{addon}} \subseteq \mathbb{P}_{\text{authen}} \subseteq \mathbb{P}_{\text{user}}$. For example, when a user assumes the role of a viewer, both the actual user and the installed add-on are restricted from commenting on or editing this resource.

C. Permission Escalation Definitions

Add-ons must adhere to the permission system enforced by team workspaces. If team workspaces fail to enforce these permission checks, add-ons would achieve permission escalation, as indicated by the red line or cross mark in Figure 2. Our work focuses on **permission escalation** issues that arise from the add-on's use of host APIs. We formalize our attack model as follows.

Definition II.1 (Permission control consistency)**.** In our definition, we let $\mathcal{O}$ represent the set of object fields (e.g., spreadsheet, document); $\mathcal{A}$ denotes a set of operations that can be applied to the object; $\mathcal{S} = \{O, E, C, V\}$ represents the five user roles, where O is short for Owner, E is short for Editor, C for Commenter and V for Viewer.

User permission representation. The user's permission set can be represented as a set of tuples $\mathbb{P}_{\text{user}} = \{p_u \mid p_u : (s_u, a_u, o_u), s_u \in \mathcal{S}, a_u \in \mathcal{A}, o_u \in \mathcal{O}\}$. Each tuple p_u refers to when the user assumes the subject role s_u, this user can operate action a_u on object o_u.

Add-on permission representation. The add-on's permissions can be represented as a set of API permission scopes: $\mathbb{P}_{\text{addon}} = \{p_a | p_a : (a, o), a \in \mathcal{A}, o \in \mathcal{O}\}$. The permission to execute each API p_a is $(action, object)$. For example, to successfully execute the API `Document.editText()`, the add-on must acquire permission to *edit* (action) the *text* (object).

Permission consistency representation. The two-level permission checking model can be represented as: $\mathbb{P}^{(U)}_{\text{addon}} \subseteq (\mathbb{P}_{\text{user}} \wedge Auth)$, where $U \subseteq \mathcal{S}$ represents the set of role the user assumes on resources. For each add-on installed by user U, the add-on permission scope is confined by user role U and cannot exceed the permission scope authorized by U, represented as $Auth$. In theory, we have $\mathbb{P}_{\text{addon}} \subseteq \mathbb{P}_{\text{authen}} \subseteq \mathbb{P}_{\text{user}} \subseteq \mathbb{P}_{\text{owner}}$.

Definition II.2 (Permission escalation)**.** We delineate three types of permission escalation scenarios. The first two scenarios stem from two-level permission checking of add-ons. The final scenario arises from resource-sharing permission checks. These three attacks are demonstrated as red crosses or red lines, as shown in Figure 2. Each of these scenarios can lead to permission escalation [7], [8].

OAuth permission escalation (E_1): $(\mathbb{P}_{\text{addon}} \not\subseteq \mathbb{P}_{\text{authen}})$, when the add-on installed on the user team workspaces can exceed the authorized permission scopes, this violates the first level permission checking and brings in permission escalation. For example, if the add-on is authorized with the `file:read` permission but is still able to use `editText()` to write to the file (green arrows in Figure 2). This results in a permission escalation.

User role permission escalation (E_2): $(\mathbb{P}_{\text{addon}} \subseteq \mathbb{P}_{\text{authen}}) \wedge (\mathbb{P}_{\text{addon}} \not\subseteq \mathbb{P}_{\text{user}})$, when an add-on installed on the user's team

TABLE I: The implementation of team workspaces and their potential permission escalation

Platform	Two-level permission system	E_1	E_2	E_3
Google Workspace	✓	○	○	○
Microsoft OneDrive	✓	○	○	●

A check mark (✓) indicates that the platform adopts the two-level permission system identified in our study. A red circle (○) signifies potential vulnerability to permission escalation. A black circle (●) signifies no vulnerability to permission escalation.

workspaces can perform actions that the user is not permitted to take, it clearly violates second-level permission checks. For example, we find that:

$$\left\{ p = (view, o) \,\middle|\, p \in \mathbb{P}^{(U)}_{\text{addon}} \wedge p \notin \mathbb{P}_{\text{viewer}} \right\}$$

The $P^{(U)}_{\text{addon}}$ refer to add-on installed by user with role U. The add-on can read some specific content of the resource whereas its installer cannot (blue arrows in Figure 2).

Sharing permission escalation (E_3): Besides permission checking when a user is assigned a role, we also consider permission sharing. As shown in Figure 2, sharing permissions are managed by the administrator, who can grant resource access to collaborators. However, if a user's assigned role is stealthily modified by add-ons, altering the sharing topology configuration $\mathbb{C}$ (purple lines in Figure 2) without administrator action, this also constitutes a permission escalation. We define this attack scenario as:

$$Addon.modify(\mathbb{C}) \rightarrow \mathbb{C}^{'}, \quad \text{where } \mathbb{C} \neq \mathbb{C}^{'}$$

Overview across representative platforms. We present an overview of the two most popular and representative team workspaces, Google Workspace and OneDrive. As shown in Table I, all of these platforms adopt a two-level permission check system. They enable user-sharing features with four types of user roles: owner, viewer, commenter, and editor. Both platforms are vulnerable to the first two permission escalation issues. However, OneDrive is robust against E_3 because it prohibits the add-ons capability from modifying user roles.

III. Overview and Challenges

Detecting the predefined three scenarios requires a thorough understanding of the permission scope for each role in the current team workspaces. However, the existing documentation on permission scope is incomplete. While previous work [4] provides partial insights into permission scopes from the user side, it does not address the scope from the add-on perspective. In this study, we systematically investigate the relevant permission scopes in different perspectives (real user vs. delegated add-on).

*a) **Real-users**:* To understand the proper permission scopes from user side, we establish five test accounts, each representing one of the five user roles in team workspaces. Subsequently, we employ a manual approach to comprehensively explore real-user roles and their associated permission scopes by assuming different roles. The exploration of the user permission scope was a one-time effort and was conducted manually. We investigate the capabilities of each user role in performing the fundamental permission groups, which will be detailed in Section IV-A.

*b) **Add-ons**:* Unlike assessing the permission scope for real users, which can be evaluated through manual review by assuming different user roles, the permission checking of the add-on involve the invocation of a large number of host APIs. The security implications of these APIs cannot be effectively analyzed without the assistance of an automated tool. So we have designed an automation tool called *TAI* that is capable of analyzing APIs, generating test cases, and automatically invoking APIs based on the official documentation [9] provided by team workspaces.

Our framework, called *TAI*, a **T**eam workspace **A**dd-on API testing tool, aims to test each API based on the official documentation provided by team workspaces. It first categorizes each API to the corresponding permission group, generates the correct test cases for each API, then execute the API under different user roles and record the execution status. Finally, we check whether the three predefined permission escalation scenarios happened. For example, if a document editing API is tested within a viewer workspace and still executes successfully, it indicates permission escalation E_2. We identified three challenges when implemented the *TAI* due to features specific to team workspaces.

- ***Challenge #1: API hierarchy.*** There is a hierarchical structure among host APIs as shown in Figure 4, different from other web-based applications that use a flat HTTP request API for resource access [10], [11]. Existing work [4], [10]–[12] on OAuth-based web applications like Slack and GitHub cannot handle this API hierarchy. In particular, host APIs have interdependent relationships, where certain APIs must be invoked before others. For example, to execute an API that edits document content, the API call chain that returns the current active document, `DocumentApp.getActiveDoc()`, must be invoked first. Therefore, the primary challenge lies in precisely capturing the hierarchical structure of host APIs.
- ***Challenge #2: Parameter generation.*** Certain APIs require context-dependent parameters for successful execution. For instance, the API `DriveApp.getFolderById(ID)` needs a valid folder *ID*. To properly invoke this API, our *TAI* must be able to supply a valid folder ID as input.
- ***Challenge #3: Efficiency.*** Team workspaces encompass a huge number of APIs, and testing every single one is neither efficient nor imperative for our objectives. Our primary goal is to identify permission escalation, necessitating the timely filtration of unnecessary APIs. For example, if a viewer lacks the capability to execute the `Document.addFooter()` API, subsequently rendering all methods reliant on the `addFooter()` meaningless for testing (e.g., `Document.addFooter().setText('footer')`), these cases should be excluded. Consequently, *TAI* is specifically designed to address this challenge, ensuring the timely filtering of unnecessary test cases.

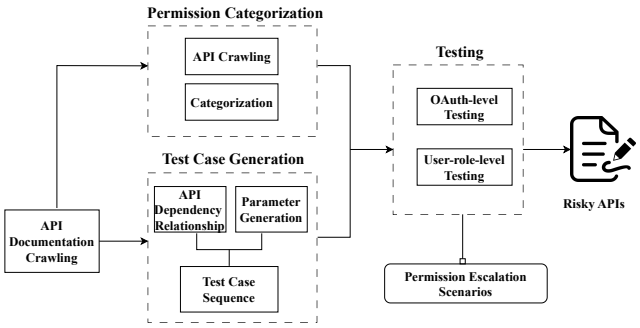

Fig. 3: Workflow of *TAI*

IV. Design of *TAI*

As shown in Figure 3, *TAI* consists of three major components: permission extraction (Section IV-A), test case generation (Section IV-B) and testing IV-C. Finally, we detail the risky APIs in Section IV-D.

A. API permission extraction

We crawl all the API documentation including the API name, its parent class, description, parameters (parameter name, type, description, etc.), tutorial code snippets (if available), and the return type as shown in Figure 5, from the official developer website of team workspaces [1]. We use Selenium [13], the most widely used web crawling framework to automate this crawling process. As shown in Figure 4, team workspaces provides a standard hierarchy for objects at varying levels of granularity. For example, the objects `Body` and `Header` are fine-grained components of the `Document` object, while `Text` is a fine-grained component of the `Body` object. We adopt this hierarchy to derive the permission tuple for each API. The total number of objects supported by each application is shown in Table II.

We leverage the API's *name*, *description*, and the pre-extracted hierarchy to infer the intended semantic permission of each API. We aim to categorize the extracted permission groups into one of the following types: $(create, object)$, $(view, object)$, $(comment, object)$, $(modify, object)$, $(delete, object)$ for later testing purpose. The `object` refers to the parent class of the API. For example, `Document.getBody()` would be categorized as $(view, \texttt{Document})$. In contrast, `Body.editAsText()` would be categorized as $(modify, \texttt{Body})$. We provide the GPT with a series of reasoning-based tutorials to enhance its chain-of-thought capabilities. Notably, team workspaces may employ diverse verbs to convey the same operation (e.g., "get a file" and "read a file", both signifying the action of viewing a file). To address this, we adopt the powerful large language model - GPT capable of capturing the semantic meaning from diverse language descriptions. GPT is instructed to assign the correct label to each API with such format *add-on, operation, object*. We use the gpt-4o-mini model [14] for this classification task. The example of the prompt we use is shown below.

[1]Google developer reference: https://developers.google.com/apps-script/reference

TABLE II: Summary of object counts

Application	Calendar	Document	Drive	Form	Gmail	Spreadsheet	Slide	**Total**
Number of Objects	7	36	6	34	6	58	47	**194**

TABLE III: Example of the returned class type of different APIs

	Example 1	**Example 2**
API	`Document.getFootnotes()`	`Document.getFooter()`
Returned type	`Footnote[]`	`FooterSection`
***TAI* strategy**	`Document.getFootnotes()[0]`	`Document.getFooter()`

You are an engineer who would like to utilize the following API.
 # Task Description
I will provide you with the API name, its description and the context of object hierarchy.
Your task is to categorize the API to one of the operation: (create, view, comment, modify or delete).
 # Output Format
[Output Format]

The OAuth permissions required for each API execution are documented in the team workspaces documentation. Currently, team workspaces provides only two types of authorization: read-only access and full access. Therefore, there is no need to match the extracted API permissions with OAuth scopes. However, the specific permissions of a subject collaborator, represented as a tuple $(collaborator, operation, object)$, are unclear. To address this, two authors independently reviewed the extracted permissions. They used the roles of viewer, commentor, and editor to verify whether the extracted permissions were actually granted to the collaborators. In total, there are 194 objects and 5$\times$ 194 permissions to be checked. The verification process took approximately eight hours to complete.

B. Test case generation

1) ***API dependency graph construction****:* We reserve the dependency between APIs (i.e., ***Challenge #1***) and construct the dependency relationship based on three components: *classes*, all *methods* (APIs) of each class, and the *return type* of each method (can be another class or void). To be detailed, we connect the *class* with its *methods* (bold arrow in Figure 4) and *method* with its *return type* (dash line in Figure 4) to construct this dependency relationship. For example, in Figure 4, `DocumentApp` has multiple methods: `create()`, `getActiveDoc()`, `openById()`, etc. These four methods return with the same type of class `Document`. Further `Document` has multiple methods: `getBody()` that return the body of `Document` and `insertText()` that helps to insert text into the `Document`.

2) ***Parameter dependency****:* Certain APIs require valid input parameters that are dependent on the outputs or responses of other APIs (i.e., ***Challenge #2***). For example, in Figure 5, the execution of `doc.setCursor(position)` (line 4 in the code snippet) requires a valid value of `position`. We refer to such parameters as context-sensitive. We categorize the API parameters into three types: context-sensitive parameters with tutorial snippets provided by team workspaces,

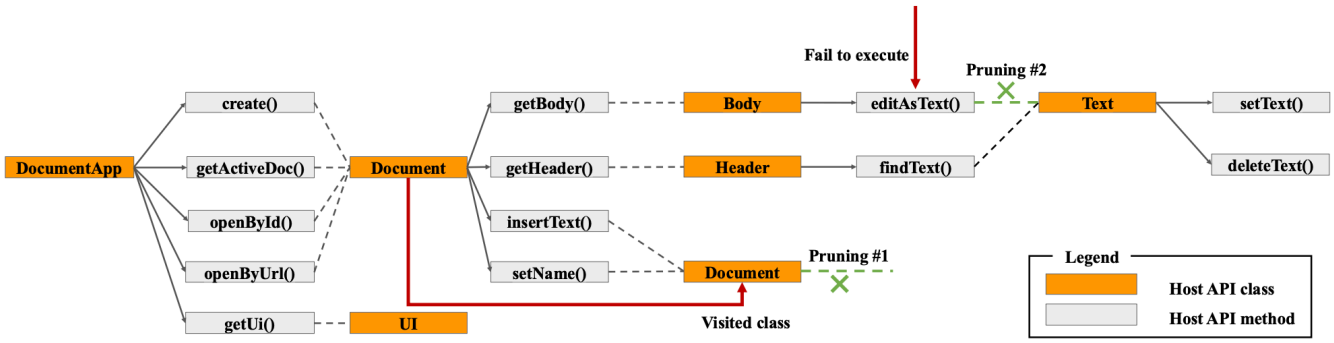

Fig. 4: An example of dependency tree of the APIs

Algorithm 1 Testing for API calls

Input G: Knowledge graph
Output C: candidates of testing $APIs$
$T \leftarrow \emptyset$, $Visited \leftarrow \emptyset$
$T \leftarrow [CalendarApp, DocumentApp, DriveApp, FormApp, GmailApp, SlidesApp, SpreadsheetApp]$
function GENERATE_TEST_CASES()
  while $T \neq \emptyset$ **do**
    $node = T.pop()$
    for all method api of $node$ in G **do**
      $Class_{type}$ = $API_{returnType}$
      if $Class_{type}$ is not visited **then** ▷ **Pruning #1**
        $T.add(Class_{type})$
        $C.add(dep : node, api)$
        $Visited \leftarrow Visited \cup Class_{type}$
function TESTING(C)
  for each candidate c in C **do**
    $response$ = Execute c
    if $response \in errors$ **then** ▷ **Pruning #2**
      filtering all APIs depend on the execution of c
    if $response \notin errors$ **then**
      record the $response$

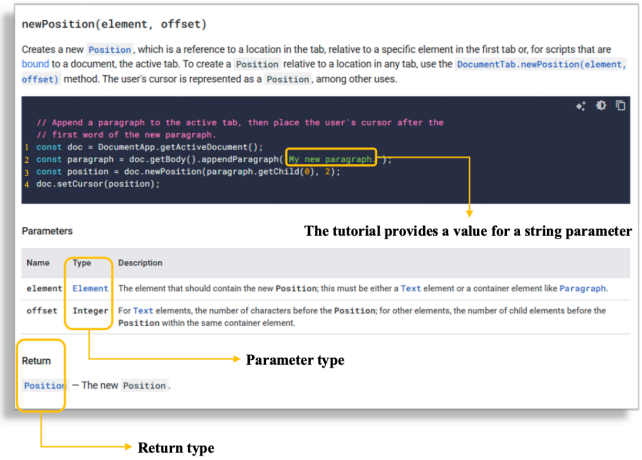

Fig. 5: Tutorial code snippets provided by team workspaces

context-sensitive parameters without tutorial snippets, and non-context-sensitive parameters.

Tutorial available: We observe that team workspaces provide detailed example code snippets for APIs requiring context related inputs, as shown in Figure 5. These code snippets are well-structured, context-sensitive, and generally do not require further modification. Consequently, we directly leverage these tutorial code snippets to capture the context parameter dependencies associated with such APIs. The only modification we apply to the tutorial code snippets is as follows: if the tutorial code snippets require `string` inputs to identify specific resources, we replace them with our customized string values, which will be detailed shortly.

Tutorial unavailable:

- **Unique class:** If team workspaces do not provide a code snippet for context-sensitive parameters, we search the dependency relationship (Figure 4) to obtain a valid parameter. Team workspaces specify the *type* of each required parameter in the official documentation, as shown in Figure 5. For each parameter with a unique class type, as illustrated in Figure 4, we traverse the connected dependency relationship to find the shortest path that leads to the required **class**. For example, `DocumentApp.getActiveDoc().getPosition()` returns a valid input for the `Position` parameter.
  If multiple paths of the same length exist, we randomly select one. If the retrieved path returns an array of the required **class**, we fetch the first element from the array, as shown in Table III. Since our purpose is to test permission escalation rather than functionality, its actual value does not make a significant difference as long as the input is valid.
- **String constant:** Specific APIs like `DocumentApp.openByName("name")` require a valid string constant for the `name` parameter. Unique `class` types can be precisely mapped through graph traversal whereas basic strings cannot. To address such cases, we maintain an attribute table of the accessed resources. This table records all the runtime values of string parameters (e.g., id, url, name).

Non context sensitive: For remaining parameters that do not fall into the previous categories, such as `Integer`, `Boolean` or context-irrelevant `String` (line 3 in Figure 5), we simply enumerate several possible values. For `Integer`, we try values like 0, 1, and 10. For cases where multiple `Integer` inputs are required and have dependencies between them, we

TABLE IV: Example of the returned class type of different APIs

Data Type	**Enumeration**
Integer	0, 1 , 5, 10
Boolean	True, False
Integer pair	GPT-generated response

query the LLM model (gpt-4o-mini) to fetch valid inputs. For example, in the API `copyFormatToRange(gridId, column, columnEnd, row, rowEnd)`, all four parameters are integers. According to the documentation of team workspaces, `column` refers to "the first column of the target range", and `columnEnd` refers to "the end column of the target range". There is an implicit dependency where `columnEnd` should be larger than `column`. We leverage the LLM model, which is capable of handling such dependencies.

3) ***Test case sequence****:* The code listing illustrated in Figure 5 demonstrates that the execution of the API `doc.setCursor(position)` (line 4) is contingent upon the successful execution of the call chain `doc = DocumentApp.getActveDocument()` (line 1), which is ***Challenge #1***. In order to generate the correct sequence of test cases, we employ a breadth-first search (**BFS**) strategy combined with a pruning mechanism. As shown in Figure 4, test case generation begins at the root class *DocumentApp* and sequentially explores each method using BFS traversal. During traversal, when we encounter a previously visited class (i.e., ***Challenge #3***), we apply a pruning mechanism (referred to as **Pruning #1**). For instance, the API `insertText()` in Figure 4 leads to a class *Document* that has already been visited, we promptly prune this branch and discontinue generating test cases along this path. Algorithm 1 shows the details of the **BFS** traversal from lines 3 - 11, and we apply the **Pruning #1** in line 8.

Due to the implicit dependencies of resource operations, such as the prerequisite creation of a file before modification, and the necessity to test deletion as the final operation in a sequence. So, our test case generation would start with APIs within the create permission scope, followed by view, comment, and modify operations, and conclude with the delete permission scope. Similarly, for sharing permission, we adhere to such design: initiating the addition of collaborators first, followed by viewing, modifying, removing collaborators, or transferring ownership.

C. Risky API testing

Due to the requirements of team workspaces, the test cases can only be tested within the add-on configuration portal, as shown in Figure 6.

1) ***Development of add-ons****:* Due to the intimate nature of add-ons, we designed and built a testing add-on that adheres to team workspace practices. We then distributed this add-on across workspaces with various user roles for testing purposes. It is worth noting that the design and development of the add-on is a one-time effort, taking approximately one hour to complete [15]. The add-on configuration portal is shown in Figure 6.

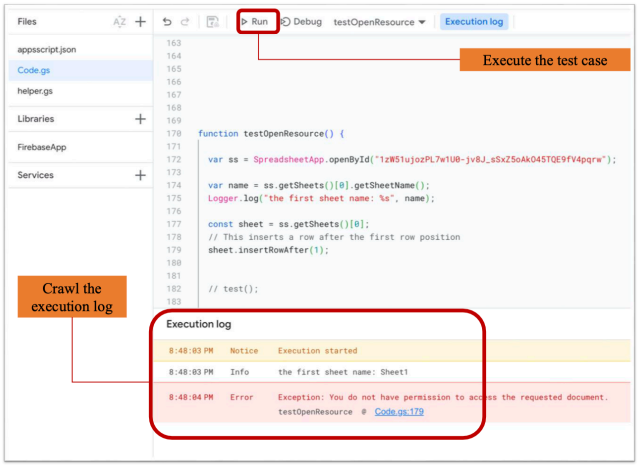

Fig. 6: The API testing phase

2) ***Test case execution****:* At the start of each testing cycle, we fetch the candidate test cases (to be tested) and update them into the project portal before execution. We use the Python library *pyautogui* [16] to simulate various hotkeys and mimic developer behavior. This allows us to paste the testing candidate into the add-on configuration page (e.g., `Code.gs` in our scenario) using the *Ctrl + V* command. *Pyautogui* also simulates the save action (*Ctrl + S*) to save the changes to the add-on project. After updating the test cases, we automated the click actions the *Run* buttons using *pyautogui* and recorded the execution log (bottom part of Figure 6) of the test case as an indicator of its execution status.

OAuth-level testing. Based on the API permission categorization results, we utilize a progressive strategy when testing OAuth-level permission escalation (E_1). We begin by granting read-only permissions to the add-ons and then test whether they can successfully execute APIs that belong to the edit or delete groups. Next, we extend the permissions to include read and edit scopes to see if the add-ons can execute APIs that belong to the delete group. All testing was conducted using an owner account.

User-role-level testing. We created five user profiles for testing. For each new round of testing, we utilized the same resource template to evaluate the execution of test cases. Each test case was tested under three different roles: viewer, commenter, and editor. Our goal is to test whether add-ons can bypass user-role-level permission checks. So the installer pre-authorizes all required permission scopes, ensuring that add-ons pass the OAuth-level check.

We monitor the execution status of each test case. If one API fails during testing (Figure 4), we proceed to prune all subsequent API calls (***Challenge #3***) that depend on the successful execution of this API, referred to as **Pruning #2**. If the API passes the testing, we document the successful execution of the API and then continue evaluating subsequent API calls that rely on this one. The details of **Pruning #2** are demonstrated in lines 12–18 of Algorithm 1.

3) ***API execution result monitor****:* Although the response from a successfully invoked API may vary, the response

TABLE V: *TAI* performance

Host App	# APIs	# Tested APIs	Potential risky APIs	Risky APIs
Calendar	218	132	0	0
Document	846	47	8	6
Drive	152	142	25	14
Gmail	167	101	0	0
Forms	418	84	2	2
Spreadsheets	1784	624	15	15
Slides	938	46	4	4

from an invalid API invocation is more consistent. After conducting an in-depth review of the team workspace developer documentation [9], we found that when the API fails to execute, the add-on will return a specific error message, such as "Exception: You do not have permission to access the requested document" and stop execution immediately. Whenever the execution returns error messages, the execution log outputs the type *Error*, as indicated in Figure 6. We mark type *Error* as the indicator of a failed execution. On the other hand, API executions that do not return any error messages are deemed successfully executed and pass the testing process.

D. Risky API identification

TAI evaluates whether the attack scenarios outlined in Section II-C are plausible by examining the runtime results of API invocations under different user roles. For example, if an API invocation that exceeds the user's permissions completes successfully without returning any errors, it suggests a potential security violation. However, verifying whether APIs designed to retrieve information return valid responses (as opposed to NULL or encrypted text) necessitates additional manual verification. We categorize the potentially risky APIs based on the criteria established in Section II-C.

Detection of E_1: If the add-on is able to execute an API that falls outside the permission scope authorized by the user, it indicates a potential security issue or permission bypass: $p_a \in \mathbb{P}_{\text{addon}} \wedge p_a \notin \mathbb{P}_{\text{authen}}$, this will be detected as attack scenario E_1.

Detection of E_2: Similarly, if the add-on is able to execute an API that operates on resources but the user cannot: $\{p_a : (a, o) \mid p_u : (s_u, a, o), p_a \in \mathbb{P}_{\text{addon}} \wedge p_u \notin \mathbb{P}_{\text{user}} \}$, this will be detected as E_2.

Detection of E_3: Finally, if add-on executes an API that modifies the current sharing configuration $\mathbb{C}$ of the resource without notifying the administrator, it will be classified as E_3.

V. Evaluation

Scope. Following the practice of previous study [4], [5], [17], we implement *TAI* and evaluate its performance on the most representative team workspaces, Google Workspace, which occupies around 73.04% market share [4].

A. Experiment setting

We collected all official documentation, totaling 4,523 APIs across seven host applications, with an average of about 646 APIs per host application. To initiate automated API testing, we created multiple testing accounts representing different user roles for collaboration. Our experiments were conducted on a series of computers assigned to different user roles to avoid any potential fingerprinting.

B. TAI *performance*

The host application email has only one user role (owner) and does not support the collaboration feature, so we focus our testing on OAuth-level permission violations for email. For the remaining six host applications, we test all three permission escalations. The specific risky APIs for each host application are listed in Table V.

*a) **Effectiveness of API categorization**:* We apply manual efforts to measure the effectiveness of *TAI*. Due to the uneven distribution of permission groups, random sampling would result in APIs belonging to the *view* or *modify* categories dominating the selection, which would not accurately reflect the performance of *TAI*. Following the methodology outlined in previous studies [10], [18], we randomly select around 200 APIs (out of 4,523, 20 APIs per permission group) and check the correctness of permission categorization. Our manual inspection identified only two problematic cases, achieving an accuracy of 99%. One such case involves the API `SlideApp.newAffineTransformBuilder()` which is categorized into *modify slide* permission group. This API returns a new `AffineTransformBuilder` that assists developers in constructing an `AffineTransform`. However, this builder does not affect the corresponding Slides unless explicitly applied to a specific slide like `AffineTransform.insertToSlide(page)`. The method `waitForAllDataExecutionsCompletion(timeoutInSeconds)` also encounters a similar issue.

*b) **Effectiveness of test case generation**:* We evaluate the performance of *TAI* during the test case generation phase, with details provided in Table VI. Our performance evaluation focuses only on APIs that are not filtered out. For APIs that are filtered out, generating test cases would be meaningless, as they will not be tested.

The majority of APIs are filtered through **Pruning #1, #2**. Out of the 1,176 APIs being tested, 482 APIs require no parameter input, while the remaining require the parameters. Among the 622 APIs that require a valid parameter, 481 have a tutorial code snippet available, and we directly apply these tutorial codes for our test case generation. For these 70 APIs requiring a valid `string` constant, we query our attribute table to fetch the corresponding string value. Around 38 APIs require other parameters like `Integer` inputs, or `Boolean` inputs.

Following the same methodology, we extracted 50 APIs from each category, for the categories *unique class type* and *others* that contain fewer than 50 APIs, we sampled all of them. This results in a total of 221 APIs. We manually evaluated the validity of *TAI*-generated test cases by executing them and observing whether any runtime errors occurred. The precision of *TAI*-generated test cases across different groups is summarized in Table VI. Our evaluation shows that only a small proportion of the *TAI*-generated test cases

TABLE VI: Test case generation

Category	# APIs	# Invalid test case (out of 50 sampled)	Precision
No Parameter Required			
No parameter	518	0	100%
Tutorial Available			
Tutorial	517	0	100%
Tutorial Unavailable			
Path dependency	33	4	87.9%
String constant	70	2	96.0%
Non context sensitive			
Others	38	0	100%

are invalid. For around 91% APIs (require no parameters or with tutorial), *TAI* achieves excellent performance - 100% accuracy. We investigate reasons for invalid test cases for the remaining three categories:

Unique class type. All four invalid test cases in Table VI are introduced by the special *Enum* parameter. For example, one API `Presentation.appendSlide(predefinedLayout)` requires the parameter of class type `predefinedLayout`. After reviewing the documentation of `predefinedLayout`, we find this is a special *Enum* class defined by team workspaces. To use it, the developer must be able to provide a predefined type like `predefinedLayout = SlidesApp.PredefinedLayout.BLANK`. However, the current design of *TAI* cannot generate valid input for the `Enum` class, as these types are not included in the dependency relationship.

String constant. The two problematic cases in this category are caused by the implicit meaning of the `string` parameter. For example, the API `Spreadsheet.getAs(contentType)` requires a string parameter `contentType`, which is not included in our pre-built attribute table. Upon investigating the documentation, we found that `contentType` must follow a specific format, as shown below:

> **contentType:** the MIME type to convert to. For most blobs, `application/pdf` is the only valid option. For images in BMP, GIF, JPEG, or PNG format, any of `image/bmp`, `image/gif`, `image/jpeg`, or `image/png` are also valid. For a Google Docs document, `text/markdown` is also valid.

In this case, *TAI* fails to provide a valid input for `contentType`.

*c) **Effectiveness of API testing**:* Although team workspaces offer a vast array of APIs, we do not need to invoke all of them for our testing. We avoid testing scenarios that are inherently safe, such as an editor executing APIs requiring edit permissions under full authorization. Despite the large number of APIs, pruning strategies (**Pruning #1**, **#2**) effectively filter out 74% of them as shown in Table V. Out of the 4,523 APIs, only 1,176 APIs would go through the risky API testing phase. The pruning strategies significantly improve the testing efficiency of *TAI* by eliminating unnecessary paths. The testing phase takes about four hours in total, with each API averaging 12 seconds to complete.

C. Large-scale scanning and findings

In Table V, the last two columns present the number of potentially problematic APIs identified by *TAI*. More specifically, *TAI* reported 54 APIs that may align with our attack scenarios. Since some APIs return null values or encrypted text that do not disclose confidential information, we conducted a manual inspection and confirmed that 41 of them pose security risks and match our criterias. For example, while the API `File.getSharingPermission()` successfully executes, it returns a `NULL` value and does not expose confidential data that the user cannot access. These false positives were filtered out through manual review.

Among the identified risky APIs, 17 pose E_2 risks, 21 pose E_3 risks, and none are classified as E_1 risks. The output of *TAI* shows that team workspaces are robust against **OAuth-level** attack scenarios (E_1). Team workspaces enforce strict OAuth-level permission check, ensuring installed add-ons do not exceed their authorized scopes. This robustness is expected. However, numerous permission escalations (E_2) arise due to inconsistent checking of the **resource user role** for both users and add-ons. Furthermore, add-on's ability to modify $\mathbb{C}$ without the admin's awareness (E_3) puts all invited collaborators at risk. We discuss the identified risky APIs that *TAI* detects and demonstrate the attacks that can be launched through several case studies.

*a) **Hidden value leakage**:* Team workspaces provide rich functionality for spreadsheets, including the ability to hide certain sheets, rows or columns. Only collaborators with *unhide* permissions can unhide and view these hidden values. However, team workspaces impose no restrictions to add-on APIs, allowing viewers without *unhide* permissions to recover hidden values via the exposed APIs. In our experiment, we concealed a *salary* column and prevented viewers from copying, downloading, or printing the spreadsheet (settings that the owner can adjust as needed). Under this setting, viewers can only view the resource content online and cannot view or recover the hidden salary column. We first tested the viewer's ability to execute `row.unhide()` API and it returned an error. However, by utilizing `row.getCell()` API, add-on installed on the viewer account can successfully fetch the hidden values even if the viewer user is forbidden from viewing the hidden values.

Even worse, team workspaces would explicitly prompt a warning notification when the owner hides a specific sheet.

> Use the *View* menu to unhide sheets. All editors of this spreadsheet can view and unhide hidden sheets.

In this case, viewers are prohibited from accessing the hidden sheets; however, by utilizing the fine-grained API `Range.getCell()`, an attacker without proper permissions (editor in this case) can recover the entire hidden sheet as shown in Figure 7.

*b) **Protected range edition**:* Additionally, we observed that spreadsheets allow the owner to set specific ranges to be protected and only editable by a specific group of people (referred to as privileged editors), rather than all editors (referred to as common editors). However, common editors are still

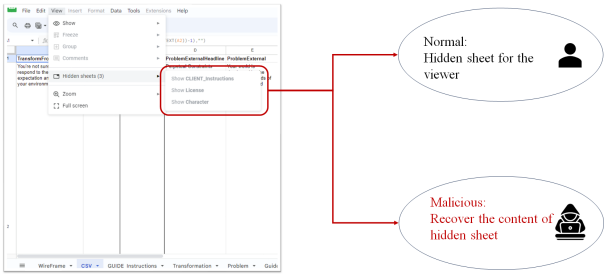

Fig. 7: Attack: hidden sheet recovery

permitted to edit the unprotected ranges. Once the protected range is *grouped*, common editors are forbidden from editing or ungrouping the whole group. However, our experiment shows that in this scenario, even though common editors are unable to either ungroup the range or edit each cell in the group, APIs exposed to add-on can still pass the execution and edit the cell value in this group. Even though the add-on fail to execute the `range.ungroup()` API.

*c) **Modification of user subject**:* The 21 APIs allow add-on to modify collaborators' subject roles without awareness of administrator. We observe that team workspaces have already banned the use of the dangerous API [19] `transferOwnership()`. Currently, transferring ownership must be done manually by the actual user rather than initiated by an add-on. To prevent misuse, team workspaces should also implement stricter management of the APIs that could be exploited by add-ons.

VI. Discussion

A. Sharing concerns

Team workspaces provide additional mechanisms for managing version history of non-native resources such as PDFs, images, and videos. It is important to note that native resources like Google Docs, Sheets, and Slides employ a different type of version history management. For non-native resources, users have the option to replace an existing file. For instance, when a user uploads a PDF file to their workspace and a file with the same name (referred to as the old version) already exists, the user can choose to replace the old version with the new one.

While the replacement option offers convenience, it poses a potential vulnerability: the sharing attributes of the old version are also copied to the new version. This introduces a security risk, especially in relation to malicious add-on. These add-ons can exploit the replace option by strategically creating a *placeholder* file with crafted names, such as *salary-report-2024.pdf*, and storing it in the victim's workspace. The add-ons can then add the attacker as a collaborator for this resource. Later, when the user uploads the actual salary file and chooses to replace the placeholder file, the attacker gains automatic access to the salary report, even if the add-on is no longer installed on the user's workspace. This scenario highlights significant security implications that must be addressed to prevent unauthorized access facilitated by the replace option. This attack is practical and has been demonstrated in previous studies [20] in other domains.

B. Countermeasures

In this section, we suggest countermeasures to help team workspaces strengthen their security against vulnerabilities.

Fine-grained permission management for add-on: Currently, team workspaces offer more stringent and fine-grained permission management for actual users. For example, spreadsheets allow for fine-grained permissions down to the row level. However, the inconsistency between strict user-level access control and the more coarse-grained add-on-level access control introduces numerous security vulnerabilities, as we have previously discussed. Team workspaces must enforce consistent and strict access control measures for add-ons to protect sensitive resources.

Strict management of sharing privilege: Team workspaces maintain strict control over resource content, but their management of resource-sharing permissions is very loose and flat. We recommend that team workspaces distribute only resource content-related permissions to collaborators, rather than resource management permissions. Besides, the sharing configuration should not be abused by add-ons. Only allowing manual operation from the user side like the discussed API `transferOwnership()` would make the resource more secure.

C. Limitations

With all the findings discovered in our paper, it is still preliminary due to the following reasons. First, our work primarily focuses on the problem of three permission escalations. Thus, *TAI* is unable to detect unseen security vulnerabilities that do not meet our criteria. Second, for the rich context parameter of API renovation, we heavily rely on tutorial code snippets provided by team workspaces. If not, we randomly choose a valid parameter value based on the dependency relationship. In the future, we can employ LLM with the dependency relationship, feed API description as prompt to generate context-sensitive parameters that can work well even without tutorials.

VII. Related Work

A. *Team workspace permission analysis.*

Previous studies mainly focus on the security and privacy of team workspaces based on manual analysis. Wan et al. investigate the interaction between different user roles and resources by manual analysis. Their analysis reveals many security violations that lead to permission escalation. Our work builds on their findings and is the first to systematically scan for problematic APIs that may lead to permission escalation in team workspaces.

Besides workspace, message systems like Slack and Microsoft Team also enable third-party applications to join as bots or delegators to invited channels and access the message history in team chat. Several studies [10], [11], [21]–[23] reveal that third-party apps in the chat system can hijack other apps, steal messages from channels that they are not invited, or perform malicious actions like merging GitHub pull requests without user's awareness. These security issues can lead to

severe impacts considering the sensitive message history and resources in the team chat system.

B. *Third-party app security.*

The security of third-party applications has been widely studied across different domains like the **Browser**, **Android**, **Internet of things** (IoT) and **Message** systems.

*a) **Browser side**:* AuthScan [24] identifies design flaws in the single-sign-on (SSO) web authentication protocol that result in seven security vulnerabilities and impacted millions of users. SSOScan [25] extends this work by developing an automated vulnerability checker for applications using single-sign-on, revealing that many top-ranked websites are susceptible to SSO vulnerabilities. Some studies [26], [27] have demonstrated that browser extensions can be malicious and have created automated tools to detect malicious extensions. Other studies [28], [29] have developed checkers for privacy policy [30] declarations, identifying numerous violations.

*b) **Android side**:* Wang et al. [31] conduct the first systematic study on mobile cross-origin risks and demonstrated that the absence of origin-based protection allows numerous attacks. Yang et al. [32] and Zhang et al. [33] attempt to identify cross-mini-app security vulnerabilities in emerging mini-apps installed on WeChat or Snapchat.

*c) **IoT side**:* Recently, researchers [34]–[36] have expanded the scope of this topic to encompass IoT platforms such as IFTTT and voice assistance devices. Bastys et al. [37] conduct the first analysis of IFTTT flows and discover numerous security vulnerabilities leading to confidential data leakage of third party applications. Mahadewa et al. [38] examine cross-app flows in IFTTT and identify many violations of privacy regulations. Additionally, SkillScanner [39] scrutinizes both the front-end and back-end code of Amazon Alexa skills, uncovering numerous privacy violations such as excessive data requests.

*d) **Message system side**:* Message systems like Slack and Microsoft Team also enable third-party applications to join as bots or delegators to invited channels and access the message history in team chat. Several studies [10], [11], [21] reveal that third-party apps in the chat system can hijack other apps, steal messages from channels that they are not invited, or perform malicious actions like merging GitHub pull requests without user's awareness. These security issues can lead to severe impacts considering the sensitive message history in the team chat system.

In contrast to previous works, the distinctive nature of team workspaces introduces new security vulnerabilities. For example, team workspaces enable multi-user collaboration under various permission levels on the same resource, which introduces fresh access control risks such as permission escalation.

C. *API analysis and testing.*

Unlike the commonly used REST APIs [10] for web applications, which are flat, APIs in team workspaces are more complex, featuring a well-structured design with interdependencies. Static analysis of such structured APIs [40] presents a challenge. Recent work such as IAceFinder [8] focuses on detecting access control inconsistencies between native (aka. C++) and Java contexts when accessing users' confidential data stored on mobile devices. IAceFinder and *TAI* share similar goals and face similar challenges. They heavily depend on call graph analysis of Android libraries to generate test cases and identify security violations. In contrast to existing work, we must address API dependencies and parameter generation while considering the unique aspects of team workspaces. To achieve this, we implemented the *TAI* to manage dependencies and efficiently prune branches as needed, expediting our testing process.

VIII. Conclusion

We conduct a comprehensive study of the access control system within team workspaces, focusing on resource management. We outline the two-level permission management framework for add-on to access sensitive resources and construct attack scenarios that lead to permission escalation, providing formal representations of these scenarios.

Utilizing the permission model established by team workspaces, we developed and implemented an automated API testing tool called *TAI*, which identifies APIs that diverge from the specified permission management protocols. *TAI* successfully identified 41 high-risk APIs and provided detailed analyses. In response to the identified security vulnerabilities, we offer countermeasures to help team workspaces mitigate these risks. We hope our analysis provides valuable insights into the security analysis of team workspaces and encourages further research in this field. Our findings serve as a wake-up call for both team workspaces and add-on developers.

Acknowledgement

This research has been partially supported by Australian Research Council Discovery Projects (DP230101196, DP240103068) and the Ministry of Education, Singapore under its Academic Research Fund Tier 3 (MOET32020-0003).

References

[1] "Google workspace marketplace," 2024. [Online]. Available: https://en.wikipedia.org/wiki/Google_Workspace_Marketplace

[2] "Apps and services," 2024. [Online]. Available: https://www.microsoft.com/en-au/microsoft-365/products-apps-services

[3] "Google workspace user stats (2023)," 2024. [Online]. Available: https://explodingtopics.com/blog/google-workspace-stats

[4] L. Wan, K. Wang, H. Wang, and G. Bai, "Is it safe to share your files? an empirical security analysis of google workspace," in *Proceedings of the ACM Web Conference 2024*, 2024.

[5] D. G. Balash, X. Wu, M. Grant, I. Reyes, and A. J. Aviv, "Security and privacy perceptions of Third-Party application access for google accounts," in *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 3397–3414. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/balash

[6] T. Bui, S. Rao, M. Antikainen, and T. Aura, "Xss vulnerabilities in cloud-application add-ons," in *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*, 2020, pp. 610–621.

[7] M. H. Meng, Q. Zhang, G. Xia, Y. Zheng, Y. Zhang, G. Bai, Z. Liu, S. G. Teo, and J. S. Dong, "Post-gdpr threat hunting on android phones: Dissecting os-level safeguards of user-unresettable identifiers." in *NDSS*, 2023.

[8] H. Zhou, H. Wang, X. Luo, T. Chen, Y. Zhou, and T. Wang, "Uncovering cross-context inconsistent access control enforcement in android," in *The 2022 Network and Distributed System Security Symposium (NDSS'22)*, 2022.

[9] "Add-ons types," 2024. [Online]. Available: https://developers.google.com/apps-script/reference/

[10] M. Zha, J. Wang *et al.*, "Hazard integrated: Understanding the security risks of app extensions on team chat systems," in *Network and Distributed Systems Security Symposium*, 2022, pp. 24–28.

[11] Y. Chen, Y. Gao, N. Ceccio, R. Chatterjee, K. Fawaz, and E. Fernandes, "Experimental security analysis of the app model in business collaboration platforms," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 2011–2028.

[12] L. Wan, K. Wang, K. T. Mahadewa, H. Wang, and G. Bai, "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*, 2024.

[13] "Selenium automates browsers. that's it!" 2024. [Online]. Available: https://www.selenium.dev/

[14] "Models: Learn about the diverse set of models that power the openai api." 2024. [Online]. Available: https://platform.openai.com/docs/models

[15] "Build google workspace add-ons," 2024. [Online]. Available: https://developers.google.com/apps-script/add-ons/how-tos/building-workspace-addons

[16] "Pyautogui," 2025. [Online]. Available: https://pypi.org/project/PyAutoGUI/

[17] H. Harkous and K. Aberer, "" 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*, 2017, pp. 127–138.

[18] C. Wang, R. Ko, Y. Zhang, Y. Yang, and Z. Lin, "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, 2023, pp. 932–944.

[19] "Api: Consent is required to transfer ownership of a file to another user [error=403]," 2024. [Online]. Available: https://issuetracker.google.com/issues/228791253

[20] N. Nikiforakis, S. Van Acker, W. Meert, L. Desmet, F. Piessens, and W. Joosen, "Bitsquatting: Exploiting bit-flips for fun, or profit?" in *Proceedings of the 22nd international conference on World Wide Web*, 2013, pp. 989–998.

[21] P. Rösler, C. Mainka, and J. Schwenk, "More is less: On the end-to-end security of group chats in signal, whatsapp, and threema," in *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2018, pp. 415–429.

[22] C. Yan, B. Guan, Y. Li, M. H. Meng, L. Wan, and G. Bai, "Understanding and detecting file knowledge leakage in gpt app ecosystem," in *Proceedings of the ACM on Web Conference 2025*, 2025, pp. 3831–3839.

[23] C. Yan, R. Ren, M. H. Meng, L. Wan, T. Y. Ooi, and G. Bai, "Exploring chatgpt app ecosystem: Distribution, deployment and security," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, 2024, pp. 1370–1382.

[24] G. Bai, J. Lei, G. Meng, S. S. Venkatraman, P. Saxena, J. Sun, Y. Liu, and J. S. Dong, "Authscan: Automatic extraction of web authentication protocols from implementations," 2013.

[25] Y. Zhou and D. Evans, "{SSOScan}: automated testing of web applications for single {Sign-On} vulnerabilities," in *23rd USENIX Security Symposium (USENIX Security 14)*, 2014, pp. 495–510.

[26] N. Jagpal, E. Dingle, J.-P. Gravel, P. Mavrommatis, N. Provos, M. A. Rajab, and K. Thomas, "Trends and lessons from three years fighting malicious extensions," in *24th USENIX Security Symposium (USENIX Security 15)*, 2015, pp. 579–593.

[27] N. Pantelaios, N. Nikiforakis, and A. Kapravelos, "You've changed: Detecting malicious browser extensions through their update deltas," in *Proceedings of the 2020 ACM SIGSAC conference on computer and communications security*, 2020, pp. 477–491.

[28] D. Bui, B. Tang, and K. G. Shin, "Detection of inconsistencies in privacy practices of browser extensions," in *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2023, pp. 2780–2798.

[29] Y. Ling, K. Wang, G. Bai, H. Wang, and J. S. Dong, "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*, 2022, pp. 1–12.

[30] C. Yan, F. Xie, M. H. Meng, Y. Zhang, and G. Bai, "On the quality of privacy policy documents of virtual personal assistant applications," *Proceedings on Privacy Enhancing Technologies*, 2024.

[31] R. Wang, L. Xing, X. Wang, and S. Chen, "Unauthorized origin crossing on mobile platforms: Threats and mitigation," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, 2013, pp. 635–646.

[32] Y. Yang, Y. Zhang, and Z. Lin, "Cross miniapp request forgery: Root causes, attacks, and vulnerability detection," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022, pp. 3079–3092.

[33] L. Zhang, Z. Zhang, A. Liu, Y. Cao, X. Zhang, Y. Chen, Y. Zhang, G. Yang, and M. Yang, "Identity confusion in {WebView-based} mobile app-in-app ecosystems," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 1597–1613.

[34] M. Surbatovich, J. Aljuraidan, L. Bauer, A. Das, and L. Jia, "Some recipes can do more than spoil your appetite: Analyzing the security and privacy risks of ifttt recipes," in *Proceedings of the 26th International Conference on World Wide Web*, 2017, pp. 1501–1510.

[35] Y. Chen, M. Alhanahnah, A. Sabelfeld, R. Chatterjee, and E. Fernandes, "Practical data access minimization in {Trigger-Action} platforms," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 2929–2945.

[36] M. M. Ahmadpanah, D. Hedin, and A. Sabelfeld, "Lazytap: On-demand data minimization for trigger-action applications," in *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2023, pp. 3079–3097.

[37] I. Bastys, M. Balliu, and A. Sabelfeld, "If this then what? controlling flows in iot apps," in *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*, 2018, pp. 1102–1119.

[38] K. Mahadewa, Y. Zhang, G. Bai, L. Bu, Z. Zuo, D. Fernando, Z. Liang, and J. S. Dong, "Identifying privacy weaknesses from multi-party trigger-action integration platforms," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2021, pp. 2–15.

[39] S. Liao, L. Cheng, H. Cai, L. Guo, and H. Hu, "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*, 2023, pp. 2321–2335.

[40] C. Yan, M. H. Meng, F. Xie, and G. Bai, "Investigating documented privacy changes in android os," *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, pp. 2701–2724, 2024.