Hazard Integrated: Understanding Security Risks in App Extensions to Team Chat Systems

Mingming Zha*[1], Jice Wang*[2], Yuhong Nan#[3], XiaoFeng Wang#[1], Yuqing Zhang#[2,4,5], and Zelin Yang[2,4]

[1]Indiana University Bloomington, {*mzha, xw7*}@*indiana.edu*
[2]National Computer Network Intrusion Protection Center,
University of Chinese Academy of Sciences, {*wangjc, zhangyq, yangzl*}@*nipc.org.cn*
[3]School of Software Engineering, Sun Yat-sen University, *nanyh@mail.sysu.edu.cn*
[4]School of Cyber Engineering, Xidian University
[5]School of Cyberspace Security, Hainan University

***Abstract*—Team Chat (*TACT*) systems are now widely used for online collaborations and project management. A unique feature of these systems is their integration of third-party apps, which extends their capabilities but also brings in the complexity that could potentially put the TACT system and its end-users at risk.**

In this paper, for the first time, we demonstrate that third-party apps in TACT systems indeed open the door to new security risks, such as privilege escalation, deception, and privacy leakage. We studied 12 popular TACT systems, following the key steps of a third-party app's life cycle (its installation, update, configuration, and runtime operations). Notably, we designed and implemented a pipeline for efficiently identifying the security risks of TA APIs, a core feature provided for system-app communication.

Our study leads to the discovery of 55 security issues across the 12 platforms, with 25 in the install and configuration stages and 30 vulnerable (or risky) APIs. These security weaknesses are mostly introduced by improper design, lack of fine-grained access control, and ambiguous data-access policies. We reported our findings to all related parties, and 8 have been acknowledged. Although we are still working with the TACT vendors to determine the security impacts of the remaining flaws, their significance has already been confirmed by our user study, which further reveals users' concerns about some security policies implemented on mainstream TACT platforms and their misconceptions about the protection in place. Also, our communication with the vendors indicates that their threat models have not been well-thought-out, with some assumptions conflicting with each other. We further provide suggestions to enhance the security quality of today's TACT systems.

I. Introduction

Team Chat (*TACT*) systems [48] (aka. business communication platforms and collaborative software) support persistent communications among a large number of individuals working on the same projects through direct messaging, group information exchange organized by public or private topics, online (web, audio or video) meetings and others. Prominent examples include Slack [60], Microsoft Teams [41], Webex Teams [10], Facebook Workplace [15] and Zoom [88]. With the surge of demands for remote collaborations, particularly under the ongoing pandemic, these systems have become hugely popular: Slack is reported to have over 10 million downloads [22], and both Microsoft Teams and Zoom are all utilized by over 100 million users around the world [21], [23], for the purposes ranging from managing government digital HQ [89], helping with commercial projects [59] to enabling remote education [39]. These systems can also be extended by integrating similar or complementary systems through *Team chat Apps* (aka. bots, integrations). For example, Slack can enrich its functionalities with over 2,200 apps, including those for Zoom, Teams, Dropbox, Gmail, Salesforce, etc. The problem is that such enrichment also introduces complexity, which coupled with loose protection on those platforms, could potentially put security-critical tasks at risk. With its importance, this issue, however, has *never* been studied before.

Shadow of security risks. Indeed, for the first time, our research shows that such Team chat Apps (called *TAs* in our research) integrated into TACT systems could punch holes in their already fragile protection, opening the door to various security risks, including privilege escalation, deception (spoofing), and privacy leak. More specifically, such a system typically implements simple role-based access control with an admin in charge of a *workspace* and the owner of each *channel* to manage the operations within the channel; an ordinary member is not supposed to interfere with other members' activities, and those outside a *private* channel do not even see its existence. Although the details of such a security model are only vaguely documented at best, its efficacy is clearly under testing in the presence of TAs. Once added to a workspace, a TA will be able to automatically perform a set of operations, such as reading or writing messages, managing

* Mingming Zha and Jice Wang are co-first authors.
# Yuhong Nan, XiaoFeng Wang and Yuqing Zhang are co-corresponding authors.

Network and Distributed Systems Security (NDSS) Symposium 2022
24-28 April 2022, San Diego, CA, USA
ISBN 1-891562-74-6
https://dx.doi.org/10.14722/ndss.2022.24387
www.ndss-symposium.org

calls, etc., based upon the *permissions* granted by its installer (e.g., an admin or a normal user). A critical question is whether these permissions remain consistent with the security model of the TA's host: e.g., whether an unprivileged member can install a TA to access the resources she is not entitled to, like messages from a private channel. Note that in the presence of the permission with an overly broad mandate, a malicious TA could look completely innocent: e.g., an app meant to get specific messages such as the posted URL links for its owner can legitimately acquire the related permission, which if not well designed might allow it to read from not only the channels the owner joins but also those he does not.

Fundamentally, the threat models of most TACT platforms, particularly their interactions with TAs, are likely not well thought out, nor are they clearly stated. As a result, not only cannot one easily understand the policies enforced on a platform and be convinced of their soundness, but she could fail to configure her workspace properly, even when some protection is indeed there to control security risks.

Security analysis on TACT platforms. To unravel this myth, we conducted the first in-depth study on 12 popular TACT platforms (i.e., Slack [60], Microsoft Teams [41], Webex Teams [10], Facebook Workplace [15], Discord [13], Mattermost [36], Bitrix24 [9], etc.), covering the key steps of a TA's lifecycle, particularly its installation, configuration, update and runtime operations. Based upon a set of common-sense security policies collected from various sources (Section III-C), our study brought to light surprising weaknesses at each of these steps, with security and privacy implications.

More specifically, due to the lack of effective TA vetting, a malicious TA can easily get installed, even without admin or other users' awareness. Making security threats of TAs particularly serious and realistic is the fact that none of the TACT platforms we analyzed properly inform their users of this security risk and related workspace settings. So there is no evidence that the workspace admin is adequately prepared to address the problem. Actually, we found that on 5 out of the 12 platforms, by default, an ordinary member can install any TA without even being aware by the admin. Particularly on Slack, an already installed TA can be stealthily updated by its owner, which enables a malicious member to add and remove dangerous capabilities unrelated to the TA's stated functionality, such as access to the whole message history of a private channel [57].

Further, a malicious TA can perform various attacks due to the problematic design across the TA's update and configuration. For example, a malicious TA can be configured to hijack the Slash Command for invoking a legitimate TA (Section IV-B); as a result, users on Slack, Teams, Webex Teams, etc., can be cheated into running malicious TAs as popular ones such as Zoom for Slack, leading to information leaks or phishing attacks. Also, the link unfurling [56] function in Slack and Facebook Workplace turns out to be a privilege escalation avenue: it allows a TA to be configured to respond to any event related to a given domain, even those from a private channel; so an unprivileged member can simply install a TA registering with a popular domain like `docs.google.com` for unauthorized reception of any URLs prefixed by that domain, such as a google doc link posted *in a private channel*. For the 12 TACT platforms investigated in our research, we manually reviewed the aforementioned TA lifecycle (i.e., installation, configuration, and update) and totally identified 25 unique instances of the above issues.

API scanning and discoveries. During the runtime stage of its lifecycle, a TA accesses resources within a workspace through TACT APIs, as regulated by different permissions, for the purposes such as making audio/video calls, viewing messages from approved channels, etc. Although such permissions are supposed to fall in line with the TA installer's privilege (e.g., only reading messages from the channels she joins), in practice, some of them may go beyond the intended scope, leading to security and privacy risks. Finding vulnerable APIs is nontrivial, as each platform we studied has tens to over a hundred APIs. Besides, an effective analysis of them requires not only setting the correct parameters for each call but also configuring the right context: e.g., the API for adding/removing call participants can only be evaluated after a call has been initiated. To address these challenges, we developed an automated tool, called *TAPIS* , to analyze a large number of APIs cross various platforms for finding those with design weaknesses. More specifically, *TAPIS* utilizes Natural Language Processing (NLP) to recover from each platform's manual parameters for each API and the interdependency among them, further construct and run call sequences inside a TA in an attempt to access the resources outside its installer's authority, so as to detect the APIs that can be exploited.

Running *TAPIS* on such TACT platforms, our study reports a total of 30 risky APIs which are used at a TA's runtime, affecting prominent systems like Slack, Webex Teams, Flock, etc. These APIs allow one to perform unauthorized operations on the channels, including disbanding a channel, removing channel members, modifying or deleting messages posted by others, and interfering with phone calls created by others. More specifically, Slack call APIs enable a channel member to update the audio/video call information posted by another member and even terminate group calls initiated by others; the APIs of Zoho Cliq (Cliq for short) allow an arbitrary user to edit or delete the messages of other channel members; a Webex Teams API allows a member to stealthily list all messages from the channel he does not join, so his access to the messages is not known to others; Twist APIs grant some of a group owner's privileges to ordinary members, such as evicting group members. Besides, in 4 out of the 12 TACT systems, an admin can freely access other users' private messages through APIs, even though this is explicitly prohibited by the platforms such as Microsoft Teams [19] [75] and perceived to be illicit by platform users (Section VII-B) and has never been documented by most platforms. More surprisingly, on the 11 platforms, a malicious TA can send spoofing URLs to a chat box via message posting APIs, which can effectively trick users into clicking innocent-looking yet malicious links.

Impacts and mitigation. We have been reporting our findings and concerns to affected parties, including Slack, Webex Teams, Cliq, and others, and are committed to helping them address all those issues. So far, both Slack and Cliq have acknowledged some problems we reported, and Slack has awarded us $1,000 bonus. Our demos, the code of *TAPIS* and other information are posted online [27]. Also discovered in our study is the lack of a clear delineation of the security policies each TACT system enforces and the misconceptions about them among the users of these systems. For example, we are surprised to find that Twist and Webex Teams grant anyone joining a channel some privileges supposed to be solely possessed by the channel owner (e.g., removing channel members), which has never been clearly explained in their documents. To understand such confusion and the impacts of our findings, we conducted a user study with over 100 TACT users. Our study not only confirms the users' serious concerns about the access control weaknesses we discovered, but also reveals the gap between their understanding of the TACT policies and those actually implemented. For example, about 74.87% of the users did not know and disagreed that their admin can access their conversations in private channels. Over 75% of the participants believed that a member should not be allowed to interfere with others' communication (calls, posting, etc.). Further from our conversation with Slack and other TACT providers, we feel that their threat models might not be clearly defined (Section VI-C). We further discuss how to enhance the security protection of these platforms.

Contributions. We summarize our contributions as follows:

• *New findings*. We conducted the first security analysis on the integration of third-party apps into popular TACT systems. Our research has led to the discovery of unexpected vulnerabilities and design weaknesses on highly popular platforms, allowing an unauthorized member to access other team members' sensitive information and interfere with their operations.

• *New understanding*. Our study also brought to light the questionable practice of popular TACT platforms that may have serious security implications. Particularly, their opaque, sometimes counterintuitive security policies and error-prone settings render their users less clear about what is under protection and what is not. Such misconception, coupled with the design weaknesses found, makes security-critical tasks running on those platforms harder to be protected than they should be.

• *Concrete step toward better protection of the TACT ecosystem*. Our research presents an opportunity to enhance the protection of the TACT ecosystem. The reported weaknesses could better inform the platform users, so actions could be taken to vet TAs to be installed, configure a workspace and manage a channel in a more secure way. Our new tool, *TAPIS* , helps automate the discovery of vulnerable APIs. TACT providers can also learn from our new understanding to better communicate with their users and improve their security models.

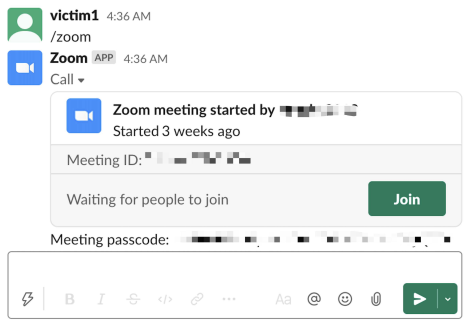

victim1 4:36 AM
/zoom
Zoom APP 4:36 AM
Call ▾
Zoom meeting started by [redacted]
Started 3 weeks ago
Meeting ID: [redacted]
Waiting for people to join
Join
Meeting passcode: [redacted]

Fig. 1. Initiate a Zoom call in Slack chat box

II. Backgroud

In this section, we introduce the background knowledge of team chat systems and team chat apps, together with their current access control design.

A. Team Chat Systems and Team Chat Apps

Like real-world organizations with shared physical workspace, a team chat system provides a virtual environment, allowing team members to collaborate online. Compared with personal messaging platforms (e.g., Facebook Messenger, KakaoTalk, and iMessage), TACT systems are characterized by two prominent features: (1) TACTs users are focusing more on group chat than one-on-one direct messaging. On a TACT platform, everyone can see the conversations that take place in a public channel. For example, on Slack [60], people from the same company are typically all members of a "workspace" (also called "teams", in other systems such as Microsoft Teams [21]). In the workspace, one can further join channels (also named "space" in Webex Teams, "group" in Facebook Workplace, respectively.) to discuss different topics or work on different projects. These channels could be either publicly accessible or made private so only invited members can participate. (2) A TACT system is explicitly designed to support collaborations, such as group video/audio chat, screen/file sharing, and project management. These features are often supported by the availability of both public and private channels and the enhancement of in-channel user interactions through integrating other collaboration tools using third-party apps (TAs, aka. integration, bots, extensions, depending on platforms). As an example shown in Figure 1, once installing a Zoom app [61] in Slack, one can directly create and initiate a Zoom call within a Slack chat box. Therefore, with the presence of other text messaging systems such as iMessage [28], WhatsApp [81], and Signal [55] that also support group chats, we consider these systems out of the scope of our research if they do not come with such unique features for team collaborations (public channels and TAs in particular).

Many TACT platforms run TA markets, in which TAs tend to be built by well-known third-party vendors (e.g., Google, Dropbox) that provide popular online services. Besides, any team member can develop and create her own TA specifically for her workspace, which may or may not be uploaded to the market. Further, depending on the setting of a TACT system, a TA can be directly installed either by the user herself or approved and installed by her workspace admin.

A TA accesses TACT resources through APIs. Such APIs are typically in the REST-API format [25], providing functionalities such as reading, writing and updating TACT system resources (e.g., message history, file, and meeting call). An API request issued by a TA is handled on the platform's server-side, which further returns a response including either the requested data (in JSON format) or a status code as feedback.

B. Access Control Models

Access control in TACT systems TACT systems implement role-based access control to protect data and other information. In most cases, there are three types of communication channels in a TACT system: *public channel, private channel, and direct message channel*[3]. For a public channel, all team members can freely join and leave the channel, and all data and resources are meant to be made public. A private channel can be created by any team member but only those invited can participate. Team members can also choose to chat via one-on-one direct messaging, which is not observable to others.

Under this model, each team member is assigned a role (e.g., the admin or the normal user), and each role is given a set of privileges to access data in the workspace. The admin has the highest privilege, capable of managing all channels created in the workspace. However, some platforms imply the role's limit in accessing the conversation within a private channel, which is in line with TACT users' perception (Section VII-B). A general user has a lower privilege, who is allowed to read and post messages in the channels she joins and edit and delete her own messages. However, she is not supposed to read, edit or delete the posts in a private channel she is not part of.

Access control on TAs. The TACT system also utilizes permission-based access control, similar to that of the Single-Sign-On system [12], to manage TAs, which leverages a unique access token and a list of permissions to control the information a TA can touch. The permission list specifies the APIs the TA can invoke. For instance, in Slack, the permission *calls:read* allows one to *view* information about ongoing and past calls through `calls.info`; *calls:write* enables an app to start and manage calls via a set of APIs such as `calls.add`. Note that such permissions can be granted to a TA either by the workspace's admin, or by the team member who installs the app, depending on the system's design and configuration.

C. Security Risks in TACT Systems

It has been reported that a Zoom vulnerability allows uninvited attendees to break in and disrupt meetings with hate-filled or pornographic content [82]. Also, its company's security practices have drawn a lot of attention – along with three lawsuits [29]. Further concerned by users are the TAs hosted in TACT systems. Miscreants are reported to move their attention from Android and iOS app stores to TA markets [70]: some Discord TAs are found to be malicious, aiming at exfiltrating sensitive data from workspaces using a key feature called webhook [68].

Given the two access control models for protecting data on a TACT platform and regulating a TA's behavior, the question becomes whether the current design and their security policies in TACT systems as well as TAs are indeed sound. Otherwise, a malicious member exploiting the weakness of a platform can put security-critical tasks running on the platform at risk. Indeed, our research shows that such security problems are present in today's TACT systems (Section IV and Section V-E), allowing unauthorized users to escalate their privilege through TAs, and further perform various attacks on other members in the same workspace.

[3]In Webex Teams, the private channel is called as moderated space [79].

TABLE I
POPULAR TACT PLATFORMS AND THE AMOUNT OF TAS AVAILABLE IN THE PLATFORMS.

Platform	Number of Downloads	Number of TAs
Slack	10M+	2200+
Microsoft Teams	100M+	700+
Webex Teams	1M+	447
Facebook Workplace	10M+	104
Flock	100,000+	53
Discord	100M+	25600+
Rocket. Chat	100,000+	118
Cliq	100,000+	56
Twist	50,000+	31
Mattermost	100,000+	29
Bitrix24	1M+	539
Zoom	100M+	1572

III. OVERVIEW OF OUR STUDY

A. The Big Picture

Platform selection. Table I summarizes the 12 TACT platforms covered by our study. To identify representative platforms, we first reviewed public reports [38], [76] for popular TACT systems. Then, we selected from those systems ones that meet the following three standards: (1) the platform supports group-based messaging with fine-grained access control (e.g., providing public and private channels for different group-chat scenarios); (2) the platform provides third-party app integration, allowing users to develop and install such apps into their systems; (3) the platform is popular with a large number of users. For example, all TACT platforms analyzed in our research have more than 100k downloads in Google Play (Table I). Through such a selection procedure, we believe our research can sufficiently cover most of the popular TACT platforms. Note that extending our study to other platforms is feasible. For example, we have proposed an automated API Scanning framework *TAPIS* (Section V), to support a more efficient discovery of security weakness in APIs across different TACT systems.

Goal and procedure. Given the 12 selected TACT platforms, our goal is to effectively detect and analyze potential security weaknesses in the integration of third-party apps on these platforms. More specifically, we aim at finding the extent to which a malicious member in the TACT system can take advantage of such apps to attack other users in the same workspace.

For this purpose, we first laid down a set of *Common Sense Security Policies* (CSSPs) for platform evaluation to overcome the opaqueness of TACT security models (Section III-C). These policies were summarized from security literature and recent public discussions (i.e., vulnerability reports of TACT platforms [26] [51]) and meant to be general across different platforms and their TAs. For example, a TA should only access resources within the scope of its installer.

Under the CSSPs, we reviewed the documentation, designs, and implementations of the 12 TACT systems to identify their potential violations of these rules. More specifically, on each of the platforms, we analyzed the whole life cycle of a TA (installation, configuration, update, and execution): our analysis started from an app's installation, update, and configuration (Section IV), looking for signals indicating security risks; further, to understand the security risks introduced during a TA's execution, particularly when it interacts with its hosting platform through APIs, we designed and implemented a framework called *TAPIS* to test whether a given API violates any CSSP. Running *TAPIS* on all 12 platforms, we were able to discover 30 risky APIs with security or privacy risks (Section VI). Lastly, we conducted a user study to understand the significance of our findings and the gap between the security policies actually implemented on these platforms and TACT users' perception.

Key findings. We summarize our key findings as follows:

(1) *TACT security risks.* Our research has led to the discovery of 55 new TACT security issues (8 has already been confirmed as vulnerabilities by TACT vendors). Examples include the "Slash Command" feature that can be hijacked for invoking a malicious TA, URL unfurl that enables an unauthorized TA to collect information from private channels (Section IV-B), lack of protection against spoofing links that can be utilized for phishing, and a set of problematic APIs that unduly grant ordinary channel members privileges to interfere with others' activities and access others' resources (Section VI).

(2) *Opaque policies and settings with security implications.* Our research shows that the security models of today's TACT platforms are often not well documented, and their security-critical settings are often unknown to users, and some of their security policies significantly deviate from user perception. For example, installing TAs on some platforms is unknown to the workspace admin by default and no warning is given to the user (Section IV-A). As another example, most of the TACT users do not know whether the admin can access their private messages, indicating that there are strong misconceptions in TACT users on certain TACT data access.

(3) *Confusing threat model.* We found that the threat models of some TACT platforms are rather confusing, as evidenced by conflicting security policies they put in place. For example, we found that Slack users in a channel are prohibited from changing or deleting each others' messages but *allowed* to edit or block each others' video/audio calls (Section VI-C).

B. Threat Model

In our research, we assume that the adversary is a workspace member with the privilege to install a third-party app into the workspace. The TA can take any action permitted under the current security model (Section II-B). Besides, we assume that the adversary performs all attacks remotely without accessing the victim's system and software. Note that the malicious TA discussed in our study may not be published at any app market. Instead, it could be directly developed and installed in the system by a normal member or the workspace admin.

C. Common Sense Security Policies

Since clear specifications of TACT security models are hard to come by, we have to collect from various sources a set of *Common-Sense Security Policies* (CSSPs) that we believe should underlie the security designs of these systems. In our research, we collected four critical CSSPs. Particularly, the first two (i.e., CSSP #1 and #2) come from the security practice of related systems (e.g., mobile computing) and the general design principle of secure systems; the other two are summarized from TACT-specific security requirements (CSSP #3) and known vulnerabilities (CSSP #4). Below we elaborate on these policies and the procedure to get them.

General security practice and principle. The security risks of third-party apps have been extensively studied in other areas (e.g., Android [17] [20], iOS [4] [80] and Chrome Extensions [32]). A common agreement among these studies is that third-party apps should be securely vetted before installation to avoid potential malicious behaviors. Also, the users should be informed of app installation and update, particularly when the app requires critical privileges or gains additional ones for resource access during this process. For example, all iOS apps need to go through a strict vetting process [4], including manual review before they become available in the Apple app store. Also, stealthy installation of third-party apps on these systems has long been considered to be a serious security risk [46].

CSSP #1: A third-party application must go through thorough security vetting during its installation and updates, and the affected users should be explicitly notified.

Besides, we believe that the least privilege principle [84] underlying almost every single secure computing system (e.g., OSes like Unix) should also be upheld by TACT platforms. For example, an ordinary Linux user is only allowed to operate on her own resources and public resources. Similarly, a TACT member is not supposed to acquire any privilege through her TA that goes beyond her roles in a workspace and a channel, and her TA should act within her privileges and not be allowed to access any resource unrelated to its functionalities.

CSSP #2: Access control on TAs should follow the least privilege principle, avoiding granting any unnecessary privilege that may bring security risks to other TACT users.

TACT-specific security requirement. TACT introduces new usage scenarios (e.g., public and private channels) compared to social messaging. Therefore, new security design choices should be made, which need to be clearly communicated to TACT users. A prominent example is the privilege of the workspace admin: our user study shows that many TACT users believe that even the admin cannot access their private chats (Section VII-B), which has been confirmed by online discussion [16] [19]. However, this security design choice has never been clearly stated in any TACT platform's documentation.

CSSP #3: TACT systems should clarify their access control design choices for sensitive user data, e.g., whether private messages can be accessed by the workspace admin.

Known vulnerabilities. Lastly, we looked into recently revealed vulnerabilities in TACT systems to find out whether any security property could be missed by these systems. In our research, we went through all vulnerabilities in Slack [26] and Rocket.Chat [51] reported by the HackerOne program [3] [50] [71]. Although these security flaws tend to violate the least privilege principle (CSSP #2), there is indeed a generic risk concerning user interfaces (UIs) that needs a new policy solution. The problem was discovered on Slack [71], which causes the discrepancy between a posted URL and its displayed text, and therfore it could be exploited for a spoofing attack. It can be addressed with the following policy:

CSSP #4: To avoid URL spoofing, TACT systems should provide adequate warning if the text of a presented link is different from the actual one it redirects.

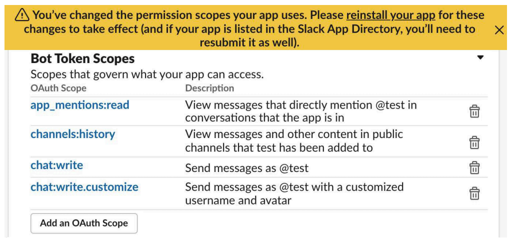
IV. Risks in App Install and Configuration

In this section, we elaborate our findings of security risks at the early stage of a TA's life cycle, including app installation, update, and configurations that could violate one or multiple CSSPs on a TACT platform.

A. Perils in App Install and Update

App install. As mentioned earlier, one can either directly install a TA from the official app market (called *public TA* in our research) or develop her own TA (called *private TA*) and install it in her workspace. This option brings security risks since our research shows that most TACT platforms today do not properly vet private TAs, allowing a malicious TA to get installed easily and affect other team members.

(1) *Install without vetting*. We found that 4 TACT systems (e.g., Slack, Twist) do not vet at all, letting any team member install private TAs on the workspace by default. While this is not the default setting for other platforms, our user study (Section VII) shows that only 21.3% of admins set "install private TA by admin" as the only option, which indicates that many admins are willing to extend this privilege to a normal user, thereby exposing their workspace to malicious TAs.



⚠ You've changed the permission scopes your app uses. Please reinstall your app for these changes to take effect (and if your app is listed in the Slack App Directory, you'll need to resubmit it as well).

Bot Token Scopes

Scopes that govern what your app can access.

OAuth Scope	Description
app_mentions:read	View messages that directly mention @test in conversations that the app is in
channels:history	View messages and other content in public channels that test has been added to
chat:write	Send messages as @test
chat:write.customize	Send messages as @test with a customized username and avatar

Fig. 2. A sample malicious app which obtains extra permissions via stealthy update.

(2) *Install without notification*. We found that 5 TACT systems allow a malicious app to be installed in their workspace without notifying affected users. Specifically, in Slack, Cliq, Twist, Webex Teams, and Flock, a TA can be installed silently without causing any notification to be issued to any workspace member.

(3) *Install by deception*. Also, our study shows that it is completely feasible for a malicious TA to masquerade as a legitimate one to deceive the admin and other team members since a private TA can easily use another TA's name and icon.

Given the weak protection mentioned above, a malicious TA can often be easily installed, which opens the door to follow-up attacks on the workspace.

App update. On all TACT platforms, a TA needs to first request permissions before accessing related resources, which is an effective way for the admin and others to be aware of the TA's capability when proper vetting is in place. However, we found that the adversary can circumvent the protection in some systems by installing a TA with harmless permissions and later updating it with more sensitive ones. It can be done since the update can be unaware to others, including the admin.

Specifically, our research shows that 5 out of the 12 TACT systems are vulnerable to this app-update attack. Slack and Twist allow an arbitrary update of an installed TA without issuing any notification. Figure 2 illustrates an example of the attack we implemented on Slack. Here, the malicious TA first claims less sensitive permissions (e.g., app_mentions:read) when installed. Then the attacker updates it and requests more sensitive permissions to receive private messages. Note that although Slack does not explicitly notify users of such an update, one can inspect the TA's main page to find all permissions requested. Even this opportunistic protection can be easily defeated: the adversary can simply update the TA twice to acquire a sensitive permission for an attack (e.g., read sensitive messages) and later drop the permission after the attack.

B. Configuration Trapdoors

When creating a TACT app, the developer can configure the TA to facilitate its interactions with its hosting TACT platform and the workspace. However, our research shows that the design of such configurations does not fully respect the TACT platform's access control model. More specifically, we found that two key settings of a TA – "Slash Command" [18],

[37], [42], [58], [72], [86], [87] and "link unfurling" [11], [40], [56], can be abused for privilege escalation: through these configurations, an ill-intended member can hijack the invocation of a benign TA so as to cheat other members into exposing their private information, or harvest the messages that she is not supposed to receive.

The root cause of the problem here is a misalignment between the security model of a TACT platform and that for controlling its TAs. Specifically, while an unprivileged team member cannot access resources in a private channel that she does not join, it is less clear what the actual permission scope of the TA she installs would be. Indeed, our research shows that in some circumstances, the permission scope of an installed TA can go beyond its installer's purview: e.g., affecting *all channels* in the workspace, including those private ones, even when the app has not been added into the channels. In this way, the TA becomes a stealthy avenue to escalate its installer's privilege. Following this line of thought, we discovered two configuration trapdoors, as follows:

Invocation hijacking via Slash Command. Slash Command [58] (aka., Command in some systems) is a mechanism that enables a member of a workspace to launch an installed TA by typing a command string in her message composer box. For example, one can invoke Zoom for Slack from her workspace to create a meeting by simply entering `"/zoom"`. For this purpose, the Zoom TA needs to claim the command `"/zoom"` on its configuration page, so it can be notified once the command is typed by someone to generate a Zoom meeting link and post it in the Slack chat box.

With its critical role in triggering an installed TA, the Slash Command has not been fully specified by these documentations of most TACT platforms, nor has it received proper protection: particularly, it is not clear at all which TA should be triggered when more than one app claims the same command. In our research, we discovered that on some platforms, including Slack, Rocket.Chat, Cliq and others, a malicious TA can strategically claim the Slash Command registered by a legitimate app in its configuration to hijack the attempt to activate the app, so as to stealthily replace the app and impersonate it to other members who may use the legitimate app. Even more seriously, in Slack, a TA installed by *anyone* can hijack the Slash Command set for *any existing TA, even when the legitimate TA has been installed and configured by the admin*. This problem, coupled with the weak TA vetting and installation control enforced on these platforms (Section IV-A), enables a malicious member to cheat anyone within the same workspace, including those inside a private channel the attacker is not part of, into disclosing their private information whenever they use the TA the attacker installs.

As an example, consider that the workspace admin has installed the Zoom TA using `"/zoom"`, so anyone in the workspace can simply enter the command to run the TA and create her meeting link posted on her public or private channel. Later a malicious member installs a malicious TA (MTA), which also claims `"/zoom"` in its configuration (which may not be observed to any member in the workspace, including the admin, see Section IV-A). Now when a member in a private channel that the attacker does not even know its existence posts a Zoom link by using `"/zoom"`, the MTA is triggered to return the meeting link the attacker controls. As a result, all the conversations (voice or text) during the online meeting could be stealthily recorded by the attacker, even though he does not have the privilege to access such communication. In addition to the information leak, the MTA can be used to launch other attacks, such as a denial of service that renders legitimate apps inaccessible through Slash Commands.

Among the 12 TACT platforms, our study has revealed that the risk is present on 5 platforms, though in three different forms: (1) Slack lets one TA seize a command already claimed, as mentioned earlier; (2) Cliq, Rocket.Chat, Bitrix24 and flock allow two TAs to claim the same command, so they will both be invoked by the command; (3) Cliq, Rocket.Chat, Bitrix24 and Flock broadcast every command to all TAs installed and those with the code for parsing and handling the command will respond. Although (2) and (3) do not seem to be particularly threatening, a malicious member can still exploit the weaknesses to escalate his privilege. Specifically, on Cliq, a TA can delete anyone's message (Section VI-C) so the attacker can draft a MTA to claim another TA's Slash Command for receiving the event triggered by the command, and then automatically delete the TA's message and post a new one. For the remaining 7 platforms, 4 do not support the Slash Command at all. The rest 3 (Zoom, Discord and Mattermost) have strict policies on using the command based TA invocation: e.g., Zoom disallows a TA to claim the Slash Command of any TA in its app market to avoid conflict, even when the app has not been installed in a specific workspace.

In addition, we also found that the Slash Command hijacking on Slack causes the privacy leak. Specifically, once a TA on Slack is invoked through a Slash Command, a notification will be sent to its server, carrying sensitive information about the user running the command, including user ID, private channel ID, and others. The exposure of such information to unauthorized parties undermines the user's privacy. For example, a MTA impersonating Zoom for Slack (by playing man-in-the-middle) can stealthily cluster the users who invoke Zoom through it, so as to find out those in the same private channel and titles of their Zoom meetings (a parameter attached to `"/zoom"`). Our video demos of this attack are posted online [27].

Message monitoring via link unfurling. Link unfurling provides a customized experience for a TA to generate previews for URLs posted by workspace members. Once a specific link is spotted by a TACT platform, the TA configured to handle the link (or domain) will receive a notification, together with the full link for it to parse. For example, if a TA registers `"docs.google.com"` as an unfurl link, it will receive any URLs starting with this prefix, then it can post the preview of the google doc to the chat box.

Similar to the Slash Command, unfurl links can also be

TABLE II
SUMMARIZED SECURITY RISKS DURING TA INSTALLATION, UPDATE AND CONFIGURATION.

Platform	App Install and Update				App Configurations	
	Installation w/o vetting	Installation w/o notification	Installation w. deception	Stealthy update	Slash Command hijacking	Message monitoring
Slack	Yes	Yes	Yes	Yes	Yes	Yes
Microsoft Teams						
Webex Teams	Yes	Yes		Yes		
Facebook Workplace*						Yes
Flock	Yes	Yes	Yes	Yes	Yes	
Discord*						
Rocket.Chat*					Yes	
Cliq		Yes	Yes	Yes	Yes	
Twist	Yes	Yes	Yes	Yes		
Mattermost*						
Bitrix24*					Yes	
Zoom						
# of affected systems	4	5	4	5	5	2

* In these TACT systems, only admin can create TA.

configured in a TA, which a malicious member can leverage to monitor all URLs related to the links posted across *all* channels, including private ones. For example, setting `"docs.google.com"` in the configuration enables a TA to receive all google docs sharing links issued by any member from any channel in the workspace. Our analysis over the 12 TACT systems shows that 2 of them have this vulnerability, including Slack and Facebook Workplace. As an example, Slack allows a TA to include no more than 5 domains for unfurling; in our research, we built a simple MTA, only 10 lines of code, to monitor all posted google docs links in a way entirely unaware to other members in the workspace. The demo of this attack is posted online [27].

C. Summary

We summarize all the identified security risks and affected TACT systems in Table II. As we can see, the problems we discovered cover all 12 systems with *25 unique instances*. We have reported these findings to all platforms. So far, Slack has acknowledged the risk of Slash Command hijacking and awarded us a bounty. Interestingly, they do not think that the link unfurling attack is a threat because all members in the workspace are considered to be "trusted", even though they put protection in place for the conversations within private channels. This indicates that their threat model has not been well defined, which we discuss in Section VI-C. Other platforms are in different stages of evaluating our reports.

V. TA API Analysis Framework - TAPIS

Unlike other stages of a TA's lifecycle (installation, update, and configuration), which can be evaluated through manual review on each TACT platform, the runtime operations of the TA entail invocation of a large number of APIs, whose security implications cannot be effectively analyzed without the help of an automated tool. In this section, we present our design of such a technique – a pipeline that enables the efficient detection of vulnerable TA APIs on different TACT platforms.

A. Motivation, Challenges and Design

Finding security risks of TA APIs is non-trivial, due to hundreds of APIs involved in each TACT system and extensive manual efforts required for analyzing such APIs. Specifically, the 12 TACT systems we studied include a total of 1,721 APIs, about 143 APIs for each platform on average. Manual analysis of all these APIs requires significant effort and also does not scale, particularly when a new security policy is introduced. Therefore, it becomes important to develop a more automated and efficient way to evaluate these APIs.

Our idea. Our framework, called *TAPIS* , is designed for analyzing a large number of TA APIs on different TACT platforms that may violate given security policies. The main idea is to automatically parse the TACT API specification from the developer guide, finding the key information (e.g., API URLs and required parameters) for triggering the API, and then evaluate whether the API can be exploited to violate a CSSP at runtime. A potential violation can be found by invoking an API in an attempt to escalate a TA owner's privilege or execute a known attack. For example, given a message-editing API, if it can be used by a normal user to revise other members' messages, we consider that this API has access control risks (violating CSSP #2 in our research).

Challenges. Although the general approach is straightforward, achieving this goal is by no means trivial. A naive solution could be simply getting all URLs and their required parameters for testing such APIs. However, this will not work in practice, and it is far from sufficient for identifying those problematic ones. Particularly, the implementation of such an API risk exploration framework faces two challenges, as elaborated below.

• *API dependency parsing.* Most TA APIs are connected with dependency relations. Specifically, one API needs to be invoked before another, and one's required parameters rely on the returned values of another API request. For example, a message delete operation depends on a valid message ID, which has to be first obtained from invoking the sending mes-

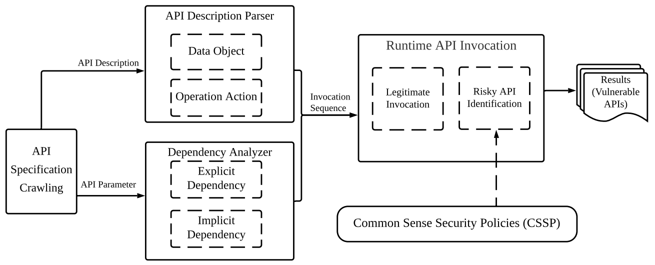


Fig. 3. Workflow of *TAPIS* .

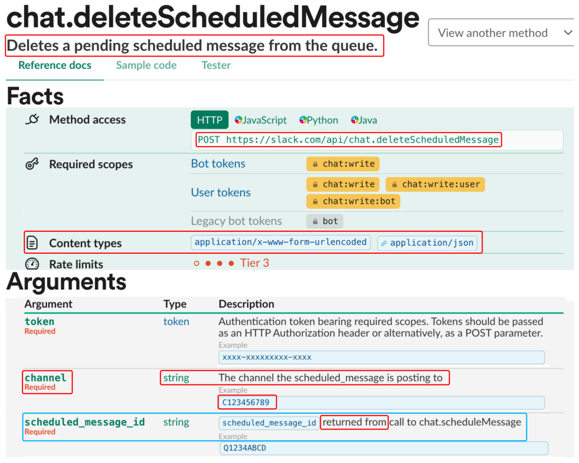
sage API. Hence, the first challenge lies in how to effectively understand the dependency relations between each API within a single platform.

• *Efficient API invocation and ordering.* An effective way to invoke each API relies on setting up a series of parameters with valid values. While such parameters can be obtained at runtime (e.g., the app's access token), the ordering of invoking each API significantly affects the efficiency of API testing. For example, the API of modifying a message should be tested before message delete. Otherwise, the message delete API will destroy the valid message ID, which is mandatory for invoking message modify.

Design of *TAPIS* . Figure 3 shows the workflow of *TAPIS* , which mainly consists of three components. Specifically, *Specification Parser* employs a set of NLP techniques to parse the TA API specifications for extracting relevant key information. Then, *Dependency Analyzer* takes the collected information and constructs the API (parameter) dependency relations as an API sequence. Ideally, the inspection of a given API needs to invoke other APIs first if it depends. Here, *TAPIS* extracts both explicit dependencies that are directly described in API specifications, as well as those implicit dependencies based on the types of such APIs (e.g., running message edit before message delete.). Lastly, *Runtime API Invocator* takes the summarized information to perform the runtime testing. It is done by first invoking the inspected API in its legitimate usage scenario and then confirming whether the API still works when the invocation violates specific CSSPs. Specifically, *TAPIS* integrates a set of mechanisms to obtain the valid parameter values required for invoking each API. For those APIs that can be effectively invoked, *TAPIS* further constructs the CSSP violation scenarios by changing specific parameter values and further determines whether the API is indeed vulnerable either automatically or with moderate human assistance.

B. API Specification Analysis

Specification crawling. *TAPIS* utilizes a platform-specific, semi-automatic approach to collect the API descriptions of TACT systems. Specifically, for each TACT platform, we manually identified the root URL of its API documentation as an input to our crawler, together with an "anchor" for the crawler to locate the API descriptions from related HTML files. As an example, the anchor for Slack is a tag with `meta name="description"` that includes the description for each API. As shown in Figure 4, for each API description, *TAPIS* collects the API URL, parameters (request type, body, etc.), their natural language descriptions, and examples when they exist. For each parameter, *TAPIS* also records whether it is mandatory or optional for the API usage (i.e., the one marked as "required"). All such information was used to not only construct a valid API call for runtime testing but also recover the dependency relation between APIs.



chat.deleteScheduledMessage

Deletes a pending scheduled message from the queue.

View another method

Reference docs Sample code Tester

Facts

Method access	HTTP JavaScript Python Java	
	`POST https://slack.com/api/chat.deleteScheduledMessage`	
Required scopes	Bot tokens	chat:write
	User tokens	chat:write chat:write:user chat:write:bot
	Legacy bot tokens	bot
Content types	application/x-www-form-urlencoded	application/json
Rate limits	○ ● ● ● Tier 3	

Arguments

Argument	Type	Description
`token` Required	token	Authentication token bearing required scopes. Tokens should be passed as an HTTP Authorization header or alternatively, as a POST parameter. Example `xxxx-xxxxxxxxx-xxxx`
`channel` Required	string	The channel the scheduled_message is posting to Example `C123456789`
`scheduled_message_id` Required	string	`scheduled_message_id` returned from call to chat.scheduleMessage Example `Q1234ABCD`

Fig. 4. A snapshot of sample API document, *TAPIS* collects texts in red squares for future processing.

Here, *TAPIS* employs Beautiful Soup [8] and Selenium [54], two widely used web crawling frameworks to crawl the API documentation. We note that while for each platform, crawling the API information would require manually assigning those interesting regions (HTML elements). It is a one-time effort and can be done within minutes. For example, among the 12 TACT systems in our research, it only takes twenty minutes for an undergraduate to locate and assign the interesting regions for web crawling.

API description parser. After collecting the related information, for each API, *TAPIS* employs NLP techniques to understand its semantics based on the API name and descriptions. More specifically, *TAPIS* summarizes the key semantic of each API as a pair that consists of two key elements– $(action, object)$. In this pair, $action$ indicates the actual operation performed on the resource (create, read, edit, and delete). $object$ refers to the data subject the API operates (e.g., message, file, calls). In other words, it describes what kind of information or service is accessed by the API. For example, given the API description *"This method retrieves public profiles of users belonging to this group"*, *TAPIS* identifies *"retrieve"* as the API action and *"profiles of users"* as the API object .

Here, *TAPIS* employs POS tagging [67] and Dependency parsing [65] to get the grammatical structure and semantic information of the API description. Both POS tagging [67] and Dependency parsing [65] are widely used NLP techniques for information retrieval. For example, previous research [45] relies on POS tagging and dependency parsing with a set of heuristics to locate privacy-related semantics. In our research, both the $action$ and the $object$ of a given API are determined based on specific dependency relations extracted from the API description. For instance, by parsing the syntax tree of the API description from root to its leaves, we first seek for verb and the noun (phrase) pair in which contained in the "Direct-object (Dobj)" relation. Here, both the identified $action$ and $object$ will further facilitate *TAPIS* to determine the optimum API invocation ordering that can more efficiently invoke APIs that required testing at runtime (Section V-D). In Appendix X-A, we present the more detailed heuristics for locating the $(action, object)$ pair based on POS tagging and Dependency parsing.

C. Dependency Analyzer

As mentioned earlier, to obtain the actual value of its mandatory parameters, the invocation of one API often relies on invoking other APIs. To this end, *TAPIS* systematically parses the parameter descriptions of each API, and finds the dependencies required for invoking each API. Here, we summarize the API dependency as two types: explicit dependency and implicit dependency.

Explicit dependency. Explicit dependency refers to those dependencies explicitly described in the parameter descriptions or reflected in the arguments and responses between two APIs. As shown in Figure 4, based on the description of parameter "scheduled_message_id", we can easily infer that the two APIs, `chat.scheduleMessage` should be invoked before `chat.deleteScheduledMessage`. To this end, for each parameter description, *TAPIS* finds keywords like "returned by", "returned from" and further locates the dependency relation between the two APIs.

Another type of explicit dependency is the case that the parameter name of one API exactly matches the API response of another (mostly in JSON format). For identifying this type of dependency, *TAPIS* first records all response keys of each API (i.e., the keys in JSON format); Then, when parsing the parameters of each API in the second round, if there is an identical string which *TAPIS* has previously seen in another API's response, it will mark the two APIs with dependency relation.

Implicit dependency. Other than those dependencies that are explicitly declared in the argument descriptions, there are certain implicit dependencies between the APIs with the same *object*s (e.g., all APIs related to file operations) based on the type of its $action$s. For example, it is mandatory to invoke the API `file.create` before invoking `file.edit`. Similarly, the API for modifying a file depends on either creating a new file, or reading an existing file in advance.

To this end, *TAPIS* classifies the $action$ of each given API into four atom operations: *create*, *read*, *modify*, and *delete*. For the same *object*, we consider operations such as *read*, *modify* and *delete* depend on its *create* API. Similarly, the operation *delete* depends on *read* API as well.

Here, since different platforms use various verbs to indicate the same operation (for example, "gets all groups", "lists all groups" both indicate to retrieve the information about groups). To better classify different $action$ terms, we manually labeled the Slack API's $action$ verbs into four categories and used these verbs as seeds to construct synonyms of the four $action$ categories using babelnet [7], a multilingual lexicalized semantic network. As a result, for each API with the same object (e.g., all APIs related to operating files), *TAPIS* specifies that an object's data operation needs to follow a temporal order of the four data operations.

D. Runtime Invocation

At runtime, *TAPIS* relies on the following two procedures to discover APIs that may contain security risks (e.g., violating the CSSPs we highlighted in Section III-C). Firstly, *TAPIS* needs to invoke the API in its legitimate usage scenario successfully. Secondly, the response information of the legitimate API invocation can further help *TAPIS* to identify whether a CSSP violation scenario indeed works (Section V-E).

Invocation mechanisms. To effectively invoke as many APIs as possible, *TAPIS* uses the following mechanisms to go across all APIs of a given platform. (1) *TAPIS* runs all APIs whose $action$ falls into the *create* category. It ensures that most of the parameters in the platform can be initialized through *create* API. In the meantime, *TAPIS* builds a lookup table, which records all the runtime values of different parameters. For example, as shown in Figure 4, by invoking `chat.scheduledMessage`, *TAPIS* will obtain the real value of the scheduled message's id and other information in the response, which might be used in other future API invocations (i.e, `chat.deleteScheduledMessage`). (2) *TAPIS* follows the temporal order of the rest of the data operations (i.e., *read*, *modify*, and *delete*) to invoke the APIs of all *object*s. During this process, *TAPIS* follows the dependency relations parsed earlier and runs all the dependent API before invoking the tested API. (3) For those mandatory parameters that do not exist in the lookup table, *TAPIS* tries different types of random values (e.g., string, integer, float, etc.) to increase the success rate of API invocation.

We note that through the above mechanisms, while most of the required parameters can be obtained automatically, *TAPIS* still needs manual specification of a few parameters as inputs, particularly those required for initiating the whole analysis process, such as the access token of the TA and the token of a TACT system user. Acquisition of such parameters is a one-time effort, which we performed in our research to bootstrap the analysis.

Identify successful invocations. While the returned response of a successfully invoked API may vary, the response of an invalid API invocation is relatively fixed. The specification of each TACT system often provides concrete examples for those invalid responses for developer debugging. To this end, for *TAPIS* , successful API invocations are the ones that do not fit any kind of invalid response. Particularly, by reviewing the documentation of all 12 TACT systems in our research, we summarized that the invalid response contains the following two types: (1) A POST API invocation may return a JSON file containing the specific error message. Notably, the term "error" is often included in this JSON. (2) Otherwise, those responses which do not return any JSON file (e.g., a GET API invocation) can be identified by their returned HTTP status code (e.g., status code 400, 403, 502). For example, status code 403 refers to the invocation is forbidden by the server [73].

E. Risky API identification

TAPIS validates whether a given CSSP can happen by changing specific parameters (as CSSP violation scenarios) and checking whether the API still works. If the API invocation under the CSSP violation scenario still returns a valid response, it very likely indicates that this API introduces security risks. Note that depending on different CSSP violation scenarios, those access control violations (i.e., CSSP #2, least privilege compliance) can be automatically confirmed based on their response, while others (e.g., CSSP #3 private message reading, and CSSP #4, URL spoofing) require extra manual inspection for its final confirmation.

Access-control violation inspection. Following the CSSP #2, we consider that a data object created by one member should not be accessed by another unless intentionally designed. Identifying this type of risk is achieved by first creating different *object*s on behave of a victim user and further performing other operations (i.e., *read*, *edit*, *delete*) of the same *object* on behalf of another (i.e., the malicious TA). For example, to test the security risk of the message's *delete* API, *TAPIS* invokes it by assigning the message id of a victim user instead of the attacker. To detail, for each TACT *object*, *TAPIS* creates one TA as the victim and another as the adversary. Then, for each *object* (e.g., file, call, etc), *TAPIS* invokes the *object*'s *create* API on behalf of the victim, meaning that the *object* belongs to a normal user in a legitimate scenario. Further, *TAPIS* invokes other data operation APIs of this *object* on behalf of the adversary. In this way, if the API still returns a valid response, we can confirm this API as risky.

Besides, we also inspect whether an adversary can operate TACT resources of other channels. In this scenario, *TAPIS* follows a similar procedure as it does for same-channel violation inspection, but with an alternative victim TA, which *object* is created in a different channel.

Private message reading by system admin. For private message reading, *TAPIS* first locates the APIs related to user messages. This is done by searching for the keyword "message" from the identified *object*s of a given TACT platform. Further, *TAPIS* invokes those APIs whose *action* are classified as *read* on behalf of the admin who created a TA for invoking the APIs, together with a message id from a private channel. If the API invocation returns a valid response, we confirm that the system admin can access private messages via the TA. Note that whether such an API is risky depends on the admin's privilege, which should be specified in the platform's documentations (whether the admin is allowed to access private messages). We discuss this issue in Section VI-C.

URL spoofing inspection. Since URL spoofing mostly happens with those APIs for sending messages, *TAPIS* inspects the same list of APIs as those in the aforementioned private message reading. Based on the reports of this vulnerability [71], URL spoofing mostly happens when parsing the texts in Markdown [33] format. To this end, *TAPIS* first invokes the message sending APIs by assigning a normal URL in Markdown format. For instance, by default, the Markdown text "www.facebook.com[www.facebook.com]" is shown as `www.facebook.com` on the user screen, and it will redirect users to the same URL once being clicked. Next, *TAPIS* alters the actual URL into a malicious one (e.g., "www.facebook.com[www.evil.com]"). Finally, we performed a manual inspection to confirm whether the URL spoofing attack indeed works. More specifically, we manually confirmed whether the user can still be directed to `www.evil.com` by clicking the on-screen text (i.e., `www.facebook.com`). Note that this manual verification step is necessary because some TACT platforms may have anti-spoofing protection in place. For example, Discord pops up an alerting window when it detects inconsistency between UI text and URL, which mitigates the risk of the URL spoofing attack.

F. Summary of TAPIS

TAPIS aims at facilitating the identification of problematic TA APIs in TACT systems. Serving this goal is its pipeline designed to significantly reduce the manual effort in building and testing each TA API. Specifically, *TAPIS* automates API testing in the following perspectives: (1) *TAPIS* automatically parses the API documentation to extract key information for generating test cases; (2) using the information, *TAPIS* automatically produces parameters for API invocations and further creates instances (i.e., the combinations of specific parameters) that may lead to violation of a given security policy.

In the meantime, we acknowledge that *TAPIS* still needs certain human interventions to complete the testing process due to the complexity of detecting problematic APIs. Particularly, to capture potential violation of CSSPs, we manually converted the policies to different testing scenarios: for example, our testing TA attempts to modify another user's resources in the same channel or access the resources in a different private channel without proper authorization (Section V-E); any success in doing so indicates a potential violation of the least privilege principle. Note that in each of such scenarios, the API to be tested and the parameters for the invocation are all automatically determined. Also, certain CSSP violations require manual confirmation: for example, we need human inspections to find out whether any warning is given to the message with URL spoofing (Section V-E).

VI. API Risk Discovery and Analysis

In this section, we elaborate on the security risks at a TA's runtime, as discovered in our study, such as the risky APIs adversaries can employ for various attacks including unauthorized access to sensitive data, denial-of-service (DoS), message spoofing, etc.

A. Experiment setting

We collected all documentations of the 12 TACT platforms, including the descriptions of 1,721 TA APIs, with an average of about 143 APIs for each TACT platform. To initialize automated API invocation, we manually opened an account for each platform, created and installed a TA, requested user authorization, and issued a POST request to obtain the access token for the TA. For example, in Zoom, one can send a POST request to `https://zoom.us/oauth/token` with the proper HTTP request headers and request body; if successful, the response body will be a JSON representation of the user's access token. All experiments in our research were run on a Macbook laptop featuring 32GB memory and an 8 core CPU.

B. TAPIS Effectiveness

Effectiveness of API semantics parser. Our manual validation shows that *TAPIS* correctly parsed the *action* and *object* of most APIs, reaching an average precision of 91.45% and 88.44%, respectively, for the 12 TACT platforms. Most false positives in parsing the *action* were caused by misidentifying the verbs in POS tagging. For example, the *action* verb in the API description "*List all IDP Groups linked to a channel*" should be "*List*." However, the POS tagging in spaCy [66] mistakenly identifies the word "*List*" as a noun. Another type of false positive was caused by the ambiguity of API description. For example, for the API description "*This method allows removing members from a given channel*", our tool mistakenly recognizes the verb "*allows*" as *action*. Such misidentification of the *action* verb also leads to misidentifying *object*. Besides, since the patterns we summarized for object identification are incomplete, some noun phrases were not accurately located in certain circumstances. For example, for the description, "*Retrieve a list of event objects*," our approach identified the *object* as "*list of objects*" instead of "*list of event objects*."

In addition, for the 12 TACT systems, *TAPIS* correctly classified 90.64% of *action* verbs into the four data operation types. To this end, we manually labeled 189 APIs from Slack and then constructed synonym sets for each operation with a total number of 929 verbs. These synonyms helped us classify the *action* verbs of the rest of 1,532 APIs in the 12 TACT system.

Effectiveness of API invocator. To identify the potential CSSP violations, as we elaborate in Section III-C, *TAPIS* does not need to invoke all APIs present in the TA API documentations, since some platforms, such as Facebook Workplace, only allow the admin to install TAs. Such a design choice naturally blocks the potential security risks of violating CSSP #2 by a normal workspace member. Thus, *TAPIS* skips most APIs for these platforms (particularly for CSSP #2). In this end, we analyzed totally 712 APIs across all platforms.

We inspected whether a given API can be effectively invoked based on the information from its return. Specifically, according to the documentation of each TACT platform, the return of an API in each TACT system has its specific feature that indicates whether its invocation is successful or not. For example, in Slack, if the API response contains a JSON with the element {`"ok":true`}, it indicates that the invocation is successful.

TABLE III
OVERALL STATISTICS OF *TAPIS* EFFECTIVENESS.

Platform	# of APIs	Success rate	Potentially problematic APIs	Confirmed problematic APIs
Slack	189	86.8%	18	6
Webex Teams	114	61.4%	5	4
Flock	11	100%	1	1
Cliq	105	80%	20	3
Twist	116	73.3%	7	7
Zoom	169	67.5%	4	1
Others	8	100%	8	8
Total	712	75.3%	63	30

Among the 712 APIs, *TAPIS* successfully invoked 536 APIs under the right context, as discovered from their dependency relations. This results in a success rate of 75.2%. The main reasons for the failures in testing the rest 176 APIs are as follows:

Firstly, most failures were caused by invalid arguments. Specifically, *TAPIS* failed to generate correct values for some parameters since no concrete example of related requests can be found in the API documentation, nor do these parameters ever show up in the returns of other APIs we can invoke. In the absence of such information, even security experts cannot determine correct values for such parameters to invoke their APIs. As an example, to call the API `SendTrialLicenseWarnMetricAck` on the Mattermost platform [35], *TAPIS* needs to provide a valid `warn_metric_id`. However, Mattermost's API documentation does not contain any specification for the parameter. Also, it cannot be found in the return values of other APIs. As a result, so far, we have not figured out how to set the parameter even manually. Another reason is lack of authorization for testing certain APIs. For example, on Slack, invocation of the log access API [63] requires a paid enterprise license granted by the Slack vendor. Finally, some invocation failures are caused by server errors out of our control, such as server-side configuration problems. For example, using the correct parameters, we still could not call Twist's get message API [74] successfully (which kept returning an error code 403) due to a server-side problem.

We note that as a framework for facilitating problematic TA API discovery, *TAPIS* does not aim at achieving guaranteed coverage. Instead, its coverage can be further improved by integrating other techniques such as NLP-based documentation analysis, REST API testing, etc.

C. Large-scale scanning and findings

Overall, it takes *TAPIS* around 40 minutes to perform its analysis over the 12 TACT platforms. In Table III, Column 4 and 5 summarize the number of potentially problematic APIs *TAPIS* identified, as well as the risky APIs confirmed by our manual inspection. Specifically, *TAPIS* reported 63

TABLE IV
SUMMARIZED RISKY TA APIS IDENTIFIED BY *TAPIS* .

Platform	API	Description	CSSP violation
Slack	POST /methods/calls.end	End a Call.	Access control
	POST /methods/calls.update	Update information about a Call.	Access control
	POST /methods/calls.participants.add	Add new participants to a Call.	Access control
	POST /methods/calls.participants.remove	Remove participants from a Call.	Access control
	POST /methods/files.sharedPublicURL	Enable a file for public/external sharing.	Access control
	POST /chat.postMessage*	Send a message to a channel.	URL spoofing
Webex Teams	GET /messages/list-messages	List all messages in a room.	Access control
	DELETE /rooms/{roomId}	Delete a room, by ID.	Access control
	PUT /rooms/{roomId}	Update details for a room, by ID.	Access control
	POST /api/v1/messages*	Post a plain text or rich text message.	URL spoofing
Microsoft Teams	GET /teams/id/channels/id/messages	Retrieve the list of messages.	Private message reading
	POST /teams/team-id/channels/channel-id/messages	Send a new chatMessage in the specified channel.	URL spoofing
Facebook Workplace	POST graph.facebook.com/app-id/subscriptions	Add new webhook subscription.	Private message reading
	POST graph.facebook.com/group-id/feed	Post into a group.	URL spoofing
Flock	POST /v1/chat.sendMessage	Allow an app to send a message to a user or a group.	URL spoofing
Discord	GET /channels/channel.id/messages	Return the messages for a channel.	Private message reading
Rocket.Chat	POST /api/v1/chat.postMessage*	Post a new chat message.	URL spoofing
Cliq	PUT /chats/{chat_id}/messages/{message_id}*	Edit a message in a conversation.	Access control
	DELETE /chats/{chat_id}messages/{message_id}*	Delete a message in a conversation.	Access control
	POST /channelsbyname/chanel-name/message	Post message in a channel.	URL spoofing
Twist	POST /groups/remove_user	Remove people from a group.	Access control
	POST /groups/remove_users	Remove a person from a group.	Access control
	GET /conversations/remove_users	Remove people from a conversation.	Access control
	GET /conversations/remove_user	Remove a person from a conversation.	Access control
	GET /api/v3/conversation_messages/get	Get the messages from a conversation.	Private message reading
	POST /api/v3/comments/add*	Adds a new comment to a thread.	URL spoofing
	POST /api/v3/conversation_messages/add*	Add a message to an existing conversation.	URL spoofing
Mattermost	POST /posts*	Create a new post in a channel.	URL spoofing
Bitrix24	POST /imconnector.send.messages	Send messages in Open Channel.	URL spoofing
Zoom	POST /v2/im/chat/messages	Send chatbot messages from chatbot app.	URL spoofing

* The API has been confirmed as a vulnerability by the platform.

APIs that may violate our highlighted CSSPs. Further, we manually inspected these APIs and confirmed 30 of them can indeed bring security risks to TACT users. The numbers in row "others" refer to APIs of the remaining 6 TACT systems. Here, most APIs in these systems were not tested because invoking these APIs require the admin privilege rather than a normal user. In our research, we only used the admin privilege to evaluate the access to private messages (Section V-E).

Among the reported 63 APIs, our manual inspection confirms that 33 APIs should not be counted as valid risky APIs because of the scenarios in which invoking these APIs does not pose any actual threats to TACT users. For example, *TAPIS* reports the API for listing users [62] as an access control violation; however, getting the list of users on the team is typically allowed by any team member.

Table IV presents the APIs that violate different CSSPs together with their detailed descriptions. Below we elaborate on the details of these risky APIs based on their functionalities.

Access control violations. Among the APIs that violate access control, Slack's call APIs allow a TA to either integrate a third-party call service (e.g., Zoom or Webex meeting) or directly use Slack's own service [64]. Through these APIs, we found that a channel member can eavesdrop on and block any private calls without authorization from the original call participants. Note that such attacks are not limited to a specific type of calls: all the call services supported by Slack are vulnerable to the unauthorized interference. More specifically, since the call id is public to all members in the same channel, an attacker could leverage the exposed id to invoke the call APIs and launch such an attack. For example, using the API `"calls.update"`, the attacker can stealthily replace a call link (e.g., a zoom vanity URL the attacker does not have access to) in the invitation issued by a channel member with her own, possibly leading to an eavesdropping attack. Also, the attacker can run `"calls.end"` to continuously disrupt the call created by other members, causing a Denial of Service attack.

In Cliq, a malicious channel member can utilize APIs to edit and delete messages sent by other members in the same channel, violating CSSP #2. So far, Cliq has acknowledged that these APIs are risky and awarded us. In Webex Teams, the API `"/v1/messages"` [78] allows an outsider to stealthily monitor the messages posted in a channel without explicitly notifying channel members. More specifically, by design in Webex Teams, anyone who joins a channel will be publicly announced to every channel member, and the user can only read chat messages from the channel that she explicitly joins. However, using the list message API through a TA, an outsider of a public channel can stealthily monitor the messages inside the channel (without being known to other channel members). This API capability is unexpected and undocumented. In addition, on multiple TACT platforms, an ordinary member is unexpectedly granted some of a channel owner's privileges. Particularly, two Webex Teams APIs (`"DELETE /rooms/roomId"` and `"PUT /rooms/roomId"`) allow the ordinary user to perform unauthorized edit, or delete of a room (channel) created by others. Also, some Twist APIs grant an ordinary member a group owner's privilege for evicting group members. None of these kind of functionalities have been properly explained to the TACT users in these platforms' documentations.

Private message reading by system admin. From the documentations of the TACT platforms analyzed in our research, we found that none of them explicitly allow the admin to read private messages. These messages should be considered off-limits to any non-channel member, including the admin, which has not only been discovered in our user study (Section VII) but also been confirmed by the TACT user's communication with Microsoft Teams [19] [75]. However, on the four platforms we investigated (including Microsoft Teams, Facebook Workplace, Discard, and Twist), we successfully accessed the user's private messages using the privileges of workspace admins. This could be a problem given the different perceptions that TACT users have, which could cause unwanted exposure of private information, based upon their false assumption about how the private messages should be protected. Besides, using the TACT system under the admin's privilege, we did not find any related function that supports the access to the user's private messages from their user interfaces. In other words, such functionality is "hidden" since it is not provided through the official UI for platform control, nor has it been properly documented. Hence, for these affected systems, we consider that the TA APIs actually open a stealthy avenue for access to private messages without any user consent, which is supported by our user study (Section VII).

URL link spoofing. Surprisingly, we discovered that 11 TACT systems have the risk that allows a malicious TA to send spoofed URLs via message posting APIs. Only Discord employs a defense strategy against the attack by disabling a URL link inconsistent with its appearance (i.e., a different URL link) on its user interface. Note that URL spoofing is considered to be a vulnerability with a significant consequence by Slack [27], which awarded us $500 for our finding that demonstrates the inadequacy of their original protection [71]. Also, three other platforms (Rocket.Chat, Webex Teams, and Twist) we informed have all acknowledged that the problem is indeed a security vulnerability [27].

VII. User Perception Analysis

Given that the TACT systems are relatively new, it is important to understand how users react to their privacy-related design choices. In this section, we elaborate on how we performed a user study to understand user perceptions of the security risks detected in our research.

A. Design of the User Study

Our study was done through Amazon Mechanical Turk (mTurk) [2], a popular task recruitment platform, under the IRB approval of our institution. To ensure the correctness of the study, we presented to the participants a set of questions to verify that they were indeed familiar with TACT systems. Specifically, we first asked them whether they had used at least one TACT platform (Q1); then, we gave them detailed questions (Q9-Q10) to confirm their TACT experience. For example, we require participants to provide the (partial) link to their workspace. Only those giving the correct answers (reviewed by our experts) were considered to be qualified participants.

TABLE V
THE USAGE RATIO OF DIFFERENT TACT SYSTEMS AMONG THE PARTICIPATES

Paltform	Usage
Microsoft Teams	68.45%
Slack	51.34%
Discord	39.57%
Webex Teams	27.27%
Facebook Workplace	16.58%
Mattermost	6.95%
Twist	5.88%
Rocket.Chat	5.35%
Cliq	3.74%
Bitrix24	3.74%
Others	7.49%

Further, we presented to the participants different data access scenarios, where each of them relates to one of our identified security issues as elaborated in Section IV and Section V. Specifically, for each data object (Slash Command, posted URLs, messages, files, calls, etc.), we described the scenario in which it is accessed by an unauthorized party. Then, we asked the participants whether they felt that their data rights were violated. For example, we showed a scenario in which the participant posts a message to his channel, and the message is later edited by Alice, another member of the channel, without his consent; we then asked the participant whether this should happen. The answer can be "Yes" or "No."

To understand the admin's perceptions of security protection on TACT, we recruited those with TACT administration experience (verifying their backgrounds with additional questions) and further sought their answers on the management of TAs. For example, we asked them how they managed TA installation (e.g., "As an administrator, how did you manage third-party app installation settings for your workspace?"). A participant's response can be "Allow any member of the workspace to install third-party apps," "Only allow certain members of the workspace to install third-party apps," "Only myself (the administrator) is allowed to install third-party apps," or "I just kept the default settings." Details of the survey questions and the participants' responses are presented on our anonymous project website [27].

B. Results and Findings

The survey study began in May 2021 and lasted two weeks. We collected 187 valid responses from 1,090 participants. The main reason for the low valid response ratio is that 659 (60.46%) participants did not finish the survey. Among the complete responses (431), many answers turned out to be incorrect, particularly those for verifying the participants' TACT experiences. For example, 244 (56.61%) submitted incorrect Workspace URLs that do not belong to any TACT system (Q9). Among the 187 valid participants, 54 of them are normal users, while the rest 133 also have admin experience.

The median age of all valid participants (110 male and 77 female) is 36. 81.28% of them hold a bachelor's or a higher degree. Table V shows the ratio of the TACT systems used by the participants.

User perceptions of access control risks. From the survey, we found that most TACT users are indeed concerned about the access control risks identified by our research. Specifically, for Questions 11 to 23 that ask the participants their perceptions of risky APIs, 80.13% of the participants agreed that accessing such data under the survey scenarios violates their privacy rights. Remarkably, the answers to Question 12 show that most (84.49%) participants agree that cross-channel data access (i.e., reading URLs) is a substantial privacy violation. Similarly, based on the collected answers from Question 14 to 24, we can see that most participants believe that even in the same channel, the data created by one member should not be accessed by another member in an unauthorized way. For example, for Question 15, 89.30% of the participants indicate that the messages created by one member should not be re-edited by another without the message creator's awareness. The overall results also reveal that the threat model TACT users perceive is actually quite different from that assumed by most TACT systems (i.e., assuming members in the same workspace or the same channel can be fully trusted).

User perceptions of private message access. The answers to Question 8 indicate that most users (74.87%) prefer their workspace admin *not* to access their private messages. In the meantime, answers to Question 25 show that 92.51% of the participants do not know the default access control settings (i.e., whether the admin can access their private messages). Given that four out of the 12 TACT systems allow the admin to read private messages (Section IV), we conclude that there is a clear gap between user perceptions and the access control design of these systems.

User perceptions of app installation. As mentioned earlier (Section IV), the default setting for TA installation in most TACT systems introduces security risks, allowing a malicious TA to be stealthily installed. In our user study, we investigated whether the TACT admins would take any action against such risks (Q5-Q7). Interestingly, our results show that only 21.38% of them choose to control TA installation fully, though most (87.04%, see Q6) know that they can, and the rest (over 76%) grant this privilege to their team members without proper vetting in place.

VIII. Discussion and Risks Mitigation

Responsible disclosure. We have reported all security issues identified in our research to the 12 vendors. As of this submission, eight of them have been officially confirmed to be security vulnerabilities by six vendors. Specifically, Slack has acknowledged the security risks of its Slash Command and URL spoofing issues and awarded us a bounty. Cliq has notified us that the problematic APIs we reported (i.e., unauthorized edit/delete a message) are indeed vulnerabilities with significant security implications, and also gives us a bounty. Rocket.Chat, Webex Teams, and Twist have all corroborated the URL spoofing vulnerabilities. Particularly, Rocket.Chat has told us that they followed our suggestion to fix the problem in their most recent releases (i.e., 3.14.6, 3.15.4, 3.16.3). Although 20 other issues we reported have not been accepted by the TACT vendors, such as Slack, our user study shows that they are indeed security concerns the TACT users care about (Section VII-B). We have not yet received any feedback about the remaining 27 issues from related vendors. A more detailed summary of the confirmed vulnerabilities can be found on our anonymous project website [27].

Limitations. With all the findings made in our research, it is still preliminary: (1) due to the challenges in data collection and vulnerability confirmation (particularly the issues unrelated to TACT APIs), our study has not covered all TACT systems in the wild; (2) *TAPIS* has made the first step toward discovering the security risks of TACT APIs with a generic framework to accommodate different security policies, but so far we have only evaluated the framework on our CSSPs. Its effectiveness in supporting other policies remains to be seen, and its automation level needs further improvement. Down the road, we will explore the techniques for enhancing *TAPIS* .

For example, the current design of TAPIS may randomly generate specific parameter values if the required information is not presented in the API documentation (Section V-D). To improve the success rate for API invocation, *TAPIS* needs better mechanisms to generate such values, instead of using those random ones.

Lessons Learned. Following are our suggestions for mitigating the security risks on TACT platforms: (1) to avoid the misconceptions as shown in our user study, TACT vendors should better define their threat models, clarifying that what is trusted in different scenarios (e.g., what actions a member of a channel is allowed to take on another channel member's message); (2) the scopes of the operations that can take place on a platform should be clearly specified (e.g., whether a private message can also be read by a privileged party) and those who performs the operations should be granted the minimum privilege.

IX. Related Work

Messaging software security. Previous works have mostly focused on the security and privacy issues not specifically related to team chat scenarios. For example, Schnitzler et al. [53] studied users' preferences for the deletion functionality in instant messengers. Particularly, they conducted their research on software such as WhatsApp, Skype, Facebook Messenger.

There are a few recent studies related to TACT systems [31], [47], [52]. However, most of them focus on individual security or privacy issues rather than a comprehensive analysis across different TACT platforms like our research. For example, Kampmann et al. [24] report a study on the delivery of phishing messages via business chat tools, in which they analyzed business chat software to identify those allowing the attacker to impersonate other members by changing their names and profile pictures and others. Ling et al. [31] performed a measurement study on Zoombombing attacks, which can intercept video and audio calls by an adversary. They found that most calls for Zoombombing come from insiders who have legitimate access to the meetings. Our research shares a similar adversary model in which the attacker is a legitimate member of the TACT system.

Third-party app security. Third-party app security has been extensively studied by previous research across different platforms. For example, prior work [14], [49], and [44] focuses on the security issues of mobile apps. Recently, researchers have extended the scope of this topic to IoT platforms, such as IFTTT [77] and voice assistant devices [85]. Particularly, Qi et al. [77] lay out the inter-rule vulnerabilities among the various third-party apps on the IFTTT markets. Ahmadpanah et al. [1] studied JavaScript-driven TAPs and discovered weaknesses that could lead to attacks on IFTTT, Zapier, and Node-RED (i.e., information leakage from unsuspecting users and usurping of the entire platform). Nan et al. [85] identified two new attack surfaces in voice assistant systems (i.e., voice squatting and voice masquerading). Compared with these works, the unique scenarios in TACT systems also raise new security and privacy issues, which have been extensively discussed in our research. For example, in TACT systems, third-party apps can be directly installed by one of the users, and its functionality may affect other members, which further raises new access control risks such as privacy leakage.

Web API analysis and testing. The TA APIs provided by TACT systems in our research follow the specification of REST API [83], which is widely used for communication across webpages and web apps. There are a number of research projects focusing on testing and analyzing the security of Web APIs. Specifically, Najib et al. [43] investigated the security performance of REST API authentication using the SHA1 and MD5 encryption algorithms in the security system of a mobile application. Tang et al. [69] summarize the REST API security strategies and mechanisms and compare the security features provided by 10 API gateway providers. Li et al. [30] modeled the REST API as a special type of Colored Petri Net and developed a tool based on the Petri Net model to describe REST APIs, which can automatically check the violations of the REST constraints. Atlidakis et al. [6] introduced an automatic, stateful fuzzing tool for REST APIs, which analyzes the Swagger specification of the REST APIs to infer dependencies among request types automatically and dynamically generates tests guided by feedback from service responses. Alberto et al. [34] implemented a testing framework using AI techniques. It leverages some testing techniques (e.g., combinatorial, search-based, and metamorphic testing) to generate test inputs by analyzing API specifications. Further, it evaluates the test outputs automatically by using patterns summarized from SUT (system under test) executions and the knowledge learned from the analysis of similar programs. Atlidakis et al. [5] designed a stateful REST API fuzzing tool, which trains a statistical model to learn common usage patterns of a target REST API, and adds small noise to generate syntactically valid and learning-based mutations. Although these approaches provide different automated mechanisms for web API testing, they cannot be directly applied to our research. Particularly, *TAPIS* is the first framework designed specifically for (1) collecting and understanding TACT system documentations with data of interests (Section V-B), and (2) constructing TACT-specific testing logic to evaluate access control violations (violations of CSSPs) through TA APIs (Section V-D) The TA API testing pipeline *TAPIS* introduced in our research complements these previous works and identifies new security and privacy issues in TACT systems.

X. Conclusion

In this paper, we report the first study over the security and privacy issues of third-party apps in different Team Chat Systems. More specifically, our research covers the lifecycle of third-party apps in TACT systems, including its installation, update, configuration, and runtime operations. The study leads to the identification of new security and privacy risks on the TACT platforms due to their design weakness, such as the misaligned access control models between the TACT system and TA, and the gap between their threat models and the risk perceptions of their users. To facilitate the identification of problematic TA APIs, we also developed a pipeline that tests a large number of APIs based on given security policies. Lastly, we offer suggestions to TACT vendors on how to mitigate security risks.

Acknowledgment

We thank all anonymous reviewers for their valuable comments and feedback to improve this paper. Jice Wang, Yuqing Zhang and Zelin Yang are supported in part by the National Natural Science Foundation of China (No. U1836210) and the Key Research and Development Science and Technology of Hainan Province (ZDYF202012). The author of Sun-yat Sen University is supported in part by the National Natural Science Foundation of China (No. 62032025).

References

[1] M. M. Ahmadpanah, D. Hedin, M. Balliu, L. E. Olsson, and A. Sabelfeld, "Sandtrap: Securing javascript-driven trigger-action platforms," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021.
[2] Amazon, "Amazon mechanical turk," https://www.mturk.com/, 2021.
[3] Andrew, "bug report," https://hackerone.com/reports/1089116, 2021.
[4] Apple, "App review," https://developer.apple.com/app-store/review/, 2021.
[5] V. Atlidakis, R. Geambasu, P. Godefroid, M. Polishchuk, and B. Ray, "Pythia: grammar-based fuzzing of rest apis with coverage-guided feedback and learning-based mutations," *arXiv preprint arXiv:2005.11498*, 2020.
[6] V. Atlidakis, P. Godefroid, and M. Polishchuk, "Restler: Stateful rest api fuzzing," in *IEEE/ACM 41st International Conference on Software Engineering (ICSE 19)*. IEEE, 2019, pp. 748–758.
[7] BabelNet, "The largest multilingual encyclopedic dictionary and semantic network," https://babelnet.org/, 2021.
[8] Beautiful Soup, "Beautiful soup documentation," https://beautiful-soup-4.readthedocs.io/en/latest/, 2021.
[9] Bitrix24, "Bitrix24," https://www.bitrix24.com, 2021.
[10] Cisco Webex, "Cisco webex," https://www.webex.com, 2021.

[11] Cliq, "Deep linking," https://www.zoho.com/cliq/help/platform/deep-linking.html, 2020.
[12] J. De Clercq, "Single sign-on architectures," in *International Conference on Infrastructure Security (InfraSec 2002)*. Springer, 2002, pp. 40–58.
[13] Discord, "Discord," https://discord.com, 2021.
[14] W. Enck, D. Octeau, P. D. McDaniel, and S. Chaudhuri, "A study of android application security." in *20th USENIX Security Symposium (USENIX Security 11)*, vol. 2, no. 2, 2011.
[15] Facebook Workplace, "Facebook workplace," https://www.workplace.com, 2021.
[16] M. Fathallah, "Chats accessible by admins?" https://techcommunity.microsoft.com/t5/microsoft-teams/chats-accessible-by-admins/m-p/103612, 2017.
[17] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android permissions demystified," in *Proceedings of the 18th ACM conference on Computer and communications security (CCS 11)*, 2011, pp. 627–638.
[18] Flock, "Flock faster with slash commands," https://blog.flock.com/flock-slash-commands, 2021.
[19] J. French, "Confidentiality, safety and secrecy of private chat in ms teams," https://techcommunity.microsoft.com/t5/microsoft-teams/confidentiality-safety-and-secrecy-of-private-chat-in-ms-teams/m-p/1067095, 2019.
[20] Google, "Publish your app," https://support.google.com/googleplay/android-developer/answer/9859751?hl=en, 2021.
[21] Google Play, "com.microsoft.teams," https://play.google.com/store/apps/details?id=com.microsoft.teams, 2021.
[22] Google Play, "com.slack," https://play.google.com/store/apps/details?id=com.Slack, 2021.
[23] Google Play, "us.zoom.videomeetings," https://play.google.com/store/apps/details?id=us.zoom.videomeetingss, 2021.
[24] M. Große-Kampmann and M. Gruber, "Business chat is confused. it hurt itself in its confusion-chishing."
[25] L. Gupta, "What is rest," https://restfulapi.net/, 2020.
[26] HackerOne, "Slack," https://hackerone.com/slack/hacktivity?type=team, 2021.
[27] HazerIntegrated, "Hazer integrated," https://sites.google.com/view/hazard-integrated, 2021.
[28] A. iMessage, "imessage," https://support.apple.com/messages, 2021.
[29] E. Institute, "Zoom faces multiple class action lawsuits over privacy complaints," https://www.expertinstitute.com/resources/insights/zoom-video-faces-multiple-class-action-suits-over-privacy-complaints, 2020.
[30] L. Li and W. Chou, "Design and describe rest api without violating rest: A petri net based approach," in *2011 IEEE International Conference on Web Services (ICWS 11)*. IEEE, 2011, pp. 508–515.
[31] C. Ling, U. Balcı, J. Blackburn, and G. Stringhini, "A first look at zoombombing," *arXiv preprint arXiv:2009.03822*, 2020.
[32] L. Liu, X. Zhang, G. Yan, S. Chen *et al.*, "Chrome extensions: Threat analysis and countermeasures." in *19th Annual Network Distributed System Security Symposium (NDSS Symposium 12)*, 2012.
[33] Mark Down, "Basic syntax," https://www.markdownguide.org/basic-syntax/#links, 2021.
[34] A. Martin-Lopez, "Ai-driven web api testing," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion 20)*, 2020, pp. 202–205.
[35] Mattermost, "Request trial license and acknowledge a warning of a metric status," https://api.mattermost.com/#operation/SendTrialLicenseWarnMetricAck, 2018.
[36] Mattermost, "Mattermost," https://mattermost.com, 2021.
[37] Mattermost, "Slash commands," https://docs.mattermost.com/developer/slash-commands.html, 2021.
[38] McAfee, "Microsoft teams: Top 10 security threats," https://www.mcafee.com/enterprise/en-us/assets/white-papers/restricted/wp-microsoft-teams-top-10.pdf, 2020.
[39] Microsoft, "Microsoft teams for education," https://www.microsoft.com/en-us/microsoft-teams/education, 2021.
[40] Microsoft Teams, "Link unfurling," https://docs.microsoft.com/en-us/microsoftteams/platform/messaging-extensions/how-to/link-unfurling, 2020.
[41] Microsoft Teams, "Microsoft teams," https://www.microsoft.com/microsoft-teams, 2021.
[42] Microsoft Teams, "Use commands in teams," https://support.microsoft.com/office/use-commands-in-teams-88f61508-284d-417f-a53d-9e082164050b, 2021.
[43] A. F. Najib, E. H. Rachmawanto, C. A. Sari, K. Sarker, N. Rijati *et al.*, "A comparative study md5 and sha1 algorithms to encrypt rest api authentication on mobile-based application," in *2019 International Conference on Information and Communications Technology (ICOIACT 19)*. IEEE, 2019, pp. 206–211.
[44] Y. Nan, M. Yang, Z. Yang, S. Zhou, G. Gu, and X. Wang, "Uipicker: User-input privacy identification in mobile applications," in *24th USENIX Security Symposium (USENIX Security 15)*, 2015, pp. 993–1008.
[45] Y. Nan, Z. Yang, X. Wang, Y. Zhang, D. Zhu, and M. Yang, "Finding clues for your secrets: Semantics-driven, learning-based privacy discovery in mobile apps." in *25th Annual Network Distributed System Security Symposium (NDSS Symposium 18)*, 2018.
[46] Norton, "The risks of third-party app stores," https://us.norton.com/internetsecurity-mobile-the-risks-of-third-party-app-stores.html, 2018.
[47] S. Oesch, R. Abu-Salma, O. Diallo, J. Krämer, J. Simmons, J. Wu, and S. Ruoti, "Understanding user perceptions of security and privacy for group chat: A survey of users in the us and uk," in *Annual Computer Security Applications Conference (ACSAC 20)*, 2020, pp. 234–248.
[48] M. H. Olson, "Remote office work: changing work patterns in space and time," *Communications of the ACM*, vol. 26, no. 3, pp. 182–187, 1983.
[49] M. Oltrogge, N. Huaman, S. Amft, Y. Acar, M. Backes, and S. Fahl, "Why eve and mallory still love android: Revisiting {TLS}(in) security in android applications," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021.
[50] M. Prins, "Eavesdropping on private slack calls," https://hackerone.com/reports/184698, 2021.
[51] Rocket.Chat, "bug report," https://hackerone.com/rocket_chat/hacktivity?type=team, 2021.
[52] 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 18)*. IEEE, 2018, pp. 415–429.
[53] T. Schnitzler, C. Utz, F. M. Farke, C. Pöpper, and M. Dürmuth, "Exploring user perceptions of deletion in mobile instant messaging applications," *Journal of Cybersecurity*, vol. 6, no. 1, p. tyz016, 2020.
[54] Selenium, "Selenium automates browsers," https://www.selenium.dev//, 2021.
[55] Signal, "Signal," https://signal.org/en/, 2021.
[56] Slack, "Unfurling links in messages," https://api.slack.com/reference/messaging/link-unfurling, 2020.
[57] Slack, "Conversation.history," https://api.slack.com/methods/conversations.history, 2021.
[58] Slack, "Enabling interactivity with slash commands," https://api.slack.com/interactivity/slash-commands, 2021.
[59] Slack, "Introducing slack connect: the future of business communication," https://slack.com/blog/transformation/slack-connect??&utm_source=hppromo&utm_medium=promo, 2021.
[60] Slack, "Slack," https://slack.com, 2021.
[61] Slack, "Slack install zoom," https://slack.com/apps/A5GE9BMQC-zoom, 2021.
[62] Slack, "Slack list users api," https://slack.com/api/users, 2021.
[63] Slack, "Slack log access api," https://slack.com/api/team.accessLogs, 2021.
[64] Slack, "Using the calls api," https://api.slack.com/apis/calls, 2021.
[65] spaCy, "Dependency parsing," https://spacy.io/usage/linguistic-features#dependency-parse, 2021.
[66] spaCy, "Industrial-strength natural language processing," https://spacy.io/, 2021.
[67] spaCy, "Part-of-speech tagging," https://spacy.io/usage/linguistic-features#pos-tagging, 2021.
[68] Steven Melendez, "Discord and slack are becoming potent tools for malware attacks," https://www.fastcompany.com/90622606/discord-and-slack-are-becoming-a-potent-tool-for-malware-attacks, 2021.
[69] L. Tang, L. Ouyang, and W.-T. Tsai, "Multi-factor web api security for securing mobile cloud," in *2015 12th International Conference on Fuzzy Systems and Knowledge Discovery (FSKD 15)*. IEEE, 2015, pp. 2163–2168.
[70] Threat Post, "Attackers blowing up discord, slack with malware," https://threatpost.com/attackers-discord-slack-malware/165295/, 2021.

[71] A. Tsunoda, "Url link spoofing," https://hackerone.com/reports/481472, 2021.
[72] Twist, "Api doc," https://developer.twist.com/v3/, 2021.
[73] Twist, "error," https://developer.twist.com/v3/#errors, 2021.
[74] Twist, "Twist get message api," https://api.twist.com/api/v3/inbox/get, 2021.
[75] H. W, "Private channels in microsoft teams are here," https://www.avepoint.com/blog/microsoft-teams/microsoft-teams-private-channels-qa, 2019.
[76] P. Wagenseil, "Zoom security issues: Here's everything that's gone wrong (so far)," *Toms guide*, pp. 1–3, 2020.
[77] Q. Wang, P. Datta, W. Yang, S. Liu, A. Bates, and C. A. Gunter, "Charting the attack surface of trigger-action iot platforms," in *Proceedings of the 26th ACM conference on Computer and communications security (CCS 19)*, 2019, pp. 1439–1453.
[78] Webex Teams, "List messages," https://developer.webex.com/docs/api/v1/messages/list-messages, 2021.
[79] Webex Teams, "Webex moderate a space," https://help.webex.com/en-us/gw1w6c/Webex-Moderate-a-Space, 2021.
[80] T. Werthmann, R. Hund, L. Davi, A.-R. Sadeghi, and T. Holz, "Psios: bring your own privacy & security to ios devices," in *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security (ASIACCS 13)*, 2013, pp. 13–24.
[81] WhatsApp, "Whatsapp," https://www.whatsapp.com, 2021.
[82] Wikipedia, "Zoombombing," https://en.wikipedia.org/wiki/Zoombombing, 2020.
[83] Wikipedia, "Overview of restful api description languages," https://en.wikipedia.org/wiki/Overview_of_RESTful_API_Description_Languages, 2021.
[84] Wikipedia, "Principle of least privilege," https://en.wikipedia.org/wiki/Principle_of_least_privilege, 2021.
[85] N. Zhang, X. Mi, X. Feng, X. Wang, Y. Tian, and F. Qian, "Dangerous skills: Understanding and mitigating security risks of voice-controlled third-party functions on virtual personal assistant systems," in *2019 IEEE Symposium on Security and Privacy (SP 19)*. IEEE, 2019, pp. 1381–1396.
[86] Zoho Cliq, "Introduction to slash commands," https://www.zoho.com/cliq/help/platform/commands.html, 2021.
[87] Zoom, "Slash commands and ui elements," https://marketplace.zoom.us/docs/guides/chatbots/slash-commands-and-ui-elements, 2021.
[88] Zoom, "Zoom," https://zoom.us, 2021.
[89] Zoom, "Zoom for government," https://zoomgov.com/, 2021.

APPENDIX

A. Extracting API Action and Object in TAPIS

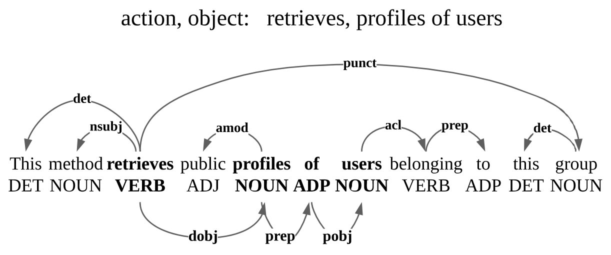

action, object: retrieves, profiles of users
punct
det
nsubj
amod
acl
prep
det
This method retrieves public profiles of users belonging to this group
DET NOUN VERB ADJ NOUN ADP NOUN VERB ADP DET NOUN
dobj
prep
pobj

Fig. 5. Dependency Parsing of a sample API description in Flock.

Figure 5 shows a sample of API description in Flock together with its parsed dependency relations, as well as the extracted $(action, object)$ pairs. In *TAPIS* , this analysis is done via the following procedures.

Action identification. In most cases, the action of a given API is a verb that holds the $root$ dependency relation to the fake root of the sentence. For example, for the API description "This method retrieves public profiles of users belonging to this group", the word "retrieves" is the main predicate verb that indicates the $action$ of the API. Therefore, *TAPIS* extracts the verb with the $root$ dependency relation as the $action$ for each API. Besides, some API descriptions use a passive sentence, for example, "This method can be used to create a new channel with the given name, image and purpose", where the $action$ word of the API follows the parse "can be used to". For such a case, if the word following "can be used to" is a verb, *TAPIS* extracts it as the $action$ of the API.

Object identification. The $object$ of API descriptions could be a none word (e.g., "user") or a none phrase (e.g., "profile of users"). We determine the $object$ of an given API based on the following dependency relations.

- **Direct-object relation (Dobj)**: The direct object of a verb is a noun phrase. Here, we firstly extract the direct object of the $action$ verb as the $object$: e.g., we firstly identify the none word "profile" as the $object$ that holds the Dobj dependency to the $action$ "retrieves" in the description "This method retrieves public profiles of users belonging to this group".
- **Noun compound modifier (compound)**: The noun compound modifier of an NP is any noun that serves to modify the head noun. To extract more accurate $object$ of the API description, if there exists a none which has the compound dependency with the Dobj word, we will identify the $object$ as "compound"+"Dobj": e.g., "message" is the direct object of the $action$ "share" in the description "Share me message into a channel". Besides, the word "me" is the noun compound modifier of the word "message", so here we extracted the phase "me message" as the $object$ for the API description.
- **Prepositional Phrase (Prep + Pobj)**: The prepositional modifier with a followed Pobj of a noun is a prepositional phrase that serves to modify the meaning of the noun. In our approach, we extract the prepositional phrase of the Dobj word, then identify the $object$ as "compound + Dobj + Prep + Pobj". For example, we extract the phrase "profiles of users" as the $object$ of the API description "This method retrieves public profiles of users belonging to this group".