Universal Cross-app Attacks: Exploiting and Securing OAuth 2.0 in Integration Platforms

Kaixuan Luo* [1], Xianbo Wang[1], Pui Ho Adonis Fung[2], Wing Cheong Lau[1], and Julien Lecomte[2]

[1]The Chinese University of Hong Kong
[2]Samsung Research America

Abstract

Integration Platforms such as Workflow Automation Platforms, Virtual Assistants and Smart Homes are becoming an integral part of the Internet. These platforms welcome third-parties to develop and distribute apps in their open marketplaces, and support "account linking" to connect end-users' app accounts to their platform account. This enables the platform to orchestrate a wide range of external services on behalf of the end-users. While OAuth is the de facto standard for account linking, the open nature of integration platforms poses new threats, as their OAuth architecture could be exploited by untrusted integrated apps.

In this paper, we examine the flawed designs of multi-app OAuth authorizations that support account linking in integration platforms. We unveil two new platform-wide attacks due to the lack of app differentiation: *Cross-app OAuth Account Takeover (COAT)* and *Request Forgery (CORF)*. As long as a victim end-user establishes account linking with a malicious app, or potentially with just a click on a crafted link, they risk unauthorized access or privacy leakage of any apps on the platform.

To facilitate systematic discovery of vulnerabilities, we develop COVScan, a semi-automated black-box testing tool that profiles varied OAuth designs to identify cross-app vulnerabilities in real-world platforms. Our measurement study reveals that among 18 popular consumer- or enterprise-facing integration platforms, 11 are vulnerable to COAT and another 5 to CORF, including those built by Microsoft, Google and Amazon. The vulnerabilities render widespread impact, leading to unauthorized control over end-users' services and devices, covert logging of sensitive information, and compromising a major ecosystem in single click (a CVE with CVSS 9.6). We responsibly reported the vulnerabilities and collaborated with the affected vendors to deploy comprehensive solutions.

*Part of this work was done during the author's internship at Samsung Research America.

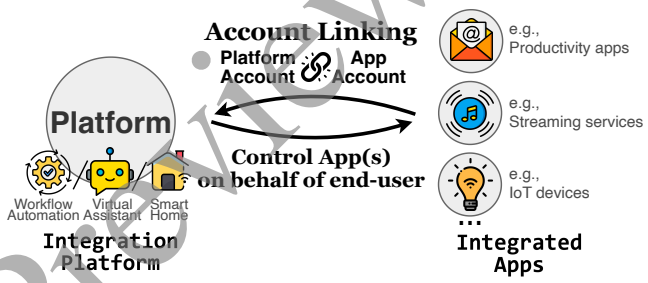

Figure 1: From Account Linking to Integrated App Control

1 Introduction

Integration Platforms. Designed to break down silos and build an interconnected ecosystem, integration platforms are cloud-based platforms that unify the functionalities of diverse integrated apps (as well as services and devices) towards a dedicated goal. In a narrow sense, they are often referred to as Workflow Automation platforms (*e.g.,* Microsoft Power Automate), which chain multiple *connectors* to trigger automated events (*e.g.,* saving Gmail attachments to Dropbox). In this paper, we also take into account other types of platforms with a similar architecture: Virtual Assistants, such as voice-based Amazon Alexa or Large Language Model (LLM)-empowered ChatGPT Plugins, allow for interactions with *voice apps* or *plugins* in natural language (*e.g.,* streaming favorite playlists on Spotify); Smart Homes, such as Google Home, aggregate device control from different *IoT providers* (*e.g.,* controlling Wi-Fi connected Philips Hue light bulbs).

Account Linking. As a fundamental feature, integration platforms support "account linking" to connect end-users' accounts at third-party apps to their platform account (as depicted in Fig. 1). Behind the scenes, apps delegate end-users' access tokens to the platform backend, primarily based on the OAuth 2.0 protocol [1] (hereafter referred to as OAuth for simplicity). This delegation empowers the platform to make API calls on behalf of the end-user, enabling aggregated access across various apps.

Our Study. *OAuth Role Reversal.* Integration platform poses a paradigm shift in OAuth roles and trust relations when it comes to account linking. In OAuth terminology, an OAuth client receives access tokens issued by an authorization server. Traditionally, OAuth clients, operated by third-party applications, register themselves at *trusted* authorization servers. In contrast, within integration platforms, it is the authorization servers – operated by *untrusted* integrated apps in the platform's open marketplace – that proactively register with the platform's OAuth client. This paper is thus motivated to explore the security considerations of OAuth within the unique context of integration platforms, endeavoring to bridge this knowledge gap.

Compromising App Accounts in Integration Platforms. In a multi-app integration architecture, an attacker can exploit the flawed OAuth design at the platform to manipulate authorizations through a malicious app. With Cross-app OAuth Account Takeover (COAT) and Cross-app OAuth Request Forgery (CORF) attacks, they respectively enable an attacker to steal a victim's authorization towards a targeted benign app, or inject their own authorization to be forcefully used by the victim. Notably, the integrated apps are most often granted with *all* and *privileged* permissions (*i.e.,* scopes in OAuth) to enable full control from the platform, be it fully automated (in workflows) or reacting to an end-user's commands (in virtual assistants and smart homes). As a result, stealing these apps' authorizations is virtually equivalent to a compromise of the whole app account (*i.e.,* account takeover). For example, being able to read, send and delete emails essentially amounts to taking over an email account.

Attacker Model: Malicious Apps. In line with the saying *One rotten apple spoils the whole barrel,* both attacks exploit the presence of a malicious app and have platform-wide implications. Their practical nature requires only that the victim be tricked into establishing account linking with an app set up by the attacker. This assumption can be further relaxed in certain circumstances, leading to single-click attacks. Moreover, the platform-wide exposure leaves all individual apps at risk with no means of self-protection.

Semi-automated Testing. We devise a decision tree that distinguishes OAuth design patterns and identifies cross-app vulnerabilities across platforms. Building on this, we develop COVScan (Cross-app OAuth Vulnerability Scanner), a semi-automated tool for scalable vulnerability detection. The tool functions by analyzing and manipulating network traffic, exempting the need to act as a malicious party or have internal knowledge of the platform. We incorporate COVScan to evaluate the security of real-world platforms, proving cost-effective and accurate compared with our manual exploitation.

Evaluation Results. Our evaluation across 18 mainstream integration platforms revealed that 16 are susceptible to cross-app attacks. Given the prevalence across different platforms and the platform-wide implications within each, this issue represents a *universal* challenge. High-profile affected platforms include Microsoft Power Automate, Amazon Alexa and Google Home, with significant security impact such as gaining unauthorized control over cloud services or physical devices, and logging sensitive information of the victim. For instance, one can take over any end-user's Microsoft 365 or Azure assets in one click, without explicit consent.

Countermeasures. The prevalence of cross-app vulnerabilities can be attributed to the fact that the latest OAuth specifications [2, 3] are unclear when analyzed from the perspective of integration platforms, which could easily lead to implementation mistakes. In response, we have proposed robust countermeasures and collaborated with affected platform vendors to deploy them, with critical severity CVE [4] assigned and $35K bug bounties awarded.

Contributions. In summary, this paper makes the following contributions:

- We demystify the authorization architecture of integration platforms, with a highlight on the trust relationship.
- We present the first in-depth analysis of cross-app OAuth attacks in integration platforms, incorporating easily met attack assumptions and real-world exploits.
- We introduce COVScan, a black-box testing tool to facilitate scalable, low-cost vulnerability detection.
- We identify severe vulnerabilities[1] in 16 out of 18 mainstream integration platforms, of which attacks in 9 platforms can be single-click, and propose robust solution to safeguard their OAuth-based account linking.

Roadmap. In the remainder of this paper, we first provide the background of OAuth and integration platforms in §2. We then identify the threat model in §3 and describe cross-app OAuth attacks in §4. Root cause and defense mechanisms are discussed in §5, followed by a comparison with similar attacks in §6. §7 dissects the methodology of vulnerability detection and seeks automation. Evaluation on real-world platforms is presented in §8. We discuss related work in §9 and conclude our work in §10. Following this we discuss ethical concerns and our responsible disclosure efforts.

Artifact Availability. The source code of COVScan can be found at https://doi.org/10.5281/zenodo.14677002.

2 Background

2.1 When OAuth Meets Integration Platforms

OAuth protocol. OAuth is a widely adopted multi-party protocol for authorization. In a typical setup, the *OAuth client* obtains an access token from an *authorization server* (AS) via the *user-agent* (usually a web browser). Then the OAuth client can access the end-user's resources with the token. Additionally, OAuth has been repurposed for authentication

[1]The attack PoCs (Proof of Concept) are available on our project website: https://mobitec.ie.cuhk.edu.hk/cross-app-oauth-security.

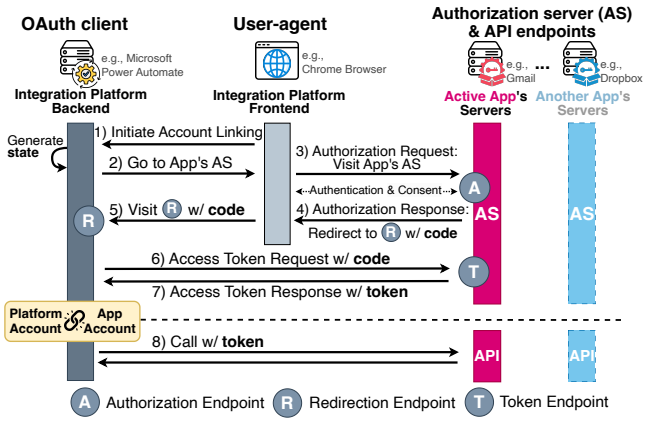

Figure 2: Protocol Flow of OAuth Authorization Code Grant in Account Linking

to provide Single Sign-On (SSO) [5–7]. In this context, OAuth client is referred to as the *relying party* (RP), and authorization server takes the role of *identity provider* (IdP).[2]

The Auth Paradigm in Integration Platforms.

Authentication. In integration platforms, authentication resembles conventional websites. Both the platform itself and the apps may employ traditional password logins or SSO to first authenticate the end-users. Establishing user sessions is the prerequisite for using the platform and account linking.

Authorization (Account Linking). In integration platforms, authorization comes into play when connecting the authenticated entities during account linking. Account linking is indispensable for apps that wish to offer a customized user experience by leveraging their existing account systems. Besides a fraction of adoptions of Basic Authentication or API keys, OAuth protocol is the most prevalent choice for account linking, which is the central focus of this paper.

Given the multi-app integration architecture, OAuth-based account linking is inherently complex. The platform provides a one-time OAuth setup for each integrated app and serves as their unified *OAuth client*. Each app registers its *authorization server* and delegates authorization to the platform backend through the end-user's interaction with platform's frontend (in a browser and/or platform's native app). The platform can then perform aggregated control over apps through API calls.

OAuth Authorization Code Grant. Zooming in on Fig. 1, Fig. 2 illustrates the detailed workflow of the OAuth authorization code grant that underlies account linking of integration platforms. When only one app is involved, the workflow is no different from traditional OAuth deployment: The end-user selects an app (*i.e.* the *active app*) to establish account linking with (Step 1), which requests the OAuth client to load pre-registered AS information from the platform backend and generate a fresh `state` parameter.[3] The end-user is then redirected to the authorization endpoint of the active app's AS by an authorization request (*e.g.,* `https://app.com/authorize?state=foo&redirect_uri=https://platform.com/redirect`,[4] Step 2&3). The end-user authenticates at the app and authorizes the account linking if they haven't done so, after which the authorization endpoint issues an authorization code (auth code) in an authorization response. The code is sent through the user-agent to the OAuth client's redirection endpoint (as specified by `redirect_uri`), which functions as a callback (*e.g.,* `https://platform.com/redirect?state=foo&code=bar`, Step 4&5).

In the next server-to-server round trip, the code (as a temporary grant) is exchanged at the token endpoint of the active app's AS for a long-lasting access `token`, to be stored by the OAuth client (Step 6&7). This wraps up the account linking process, associating the access token which indicates the app account's access with the end-user's platform account. From this point, the platform can call APIs with the `token` attached to control the app on behalf of the end-user (Step 8).

To clarify, our study does not encompass OAuth for public clients prevalent in native apps' use cases [11]. Account linking solely involves confidential clients in web servers, which holds true also for mobile integration platforms such as voice-based virtual assistants and smart homes. In these scenarios, access tokens retrieved are not consumed by the platform's mobile client (as in public clients), but instead associated with end-user's platform account by the platform backend.

2.2 App Integration Process

Apps are pre-registered, developed at integration platform's developer console, and distributed before they can be utilized.

Registration. Besides basic information such as app name and icon, the most critical aspect of registration is account linking information. Each app registers its authorization endpoint URL, token endpoint URL, `client_id`, `client_secret`, *etc.* at the platform. These fields are usually entered as static entries, but some platforms allow dynamic code to give apps greater control over how their account linking functions, such as customizing access token requests with JavaScript.[5]

The platform stores the registered information at its backend as the app's manifest and issues each app a unique identifier (*i.e.,* an app ID) to better manage its transactions internally. The platform also assigns each app a `redirect_uri`, which can be either distinct or universal across apps.

[2]This work predominantly uses the terms *OAuth client* and *authorization server* to align with the paper's focus on authorization. Note that the OAuth-based OpenID Connect (OIDC) protocol [8], which is designed specifically for SSO purposes, is neither related to nor supported by any integration platforms for account linking, making it out of scope.

[3]An unforgeable value, potentially JWT (JSON Web Token) [9]-encoded [10], to maintain state information in OAuth.

[4]For URL samples in this paper, several OAuth parameters like `response_type`, `scope` and `grant_type` are omitted for simplicity.

[5]This has an impact on the defense of our attacks, detailed in Appendix A.

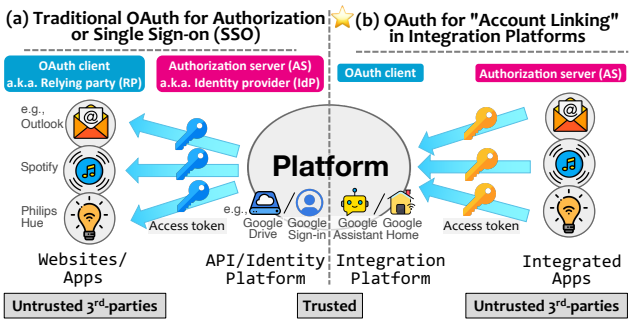

Figure 3: Flipped Paradigm: OAuth Role Reversal

Development. While core functionalities are handled by external APIs, apps must implement essential business logic at the platform so that it can invoke their APIs when appropriate and in a specified manner. These API wrappers are known as *triggers* and *actions* in workflow automation platforms, and as *intents* in virtual assistants and smart homes. Apps in virtual assistants may further extend the API responses to provide audio or visual responses to end-users.

Distribution. After registration and development, apps can be distributed through varied channels: The standard approach is to publish in the open marketplace after vetting. Some platforms also support a sharing-based, targeted distribution channel without vetting, allowing installation as a custom app via a private link or invitation. For example, app distribution within the same organization does not require vetting in Microsoft Power Automate and Slack workflow builder.

2.3 Paradigm Shift with OAuth Role Reversal

Although based on the established OAuth flow (Fig. 2), account linking in integration platform presents a unique paradigm shift in OAuth roles compared to traditional OAuth, with *ASes now becoming untrusted*, as depicted in Fig. 3:

In a traditional OAuth scenario (Fig. 3a), well-known "platform" like Google serves as the AS for third-party apps. The third-parties, such as Outlook and Spotify, act as OAuth clients, because they rely on the resources (*e.g.,* Google Drive access) or identities ("Sign in with ...") provided by the trustworthy AS. These third-parties, commonly known as OAuth apps [12], can be registered freely at the AS, making them untrusted.

However, in account linking for integration platforms (Fig. 3b), there is a role reversal when adapting the traditional OAuth framework: Well-known entities like Google now run the OAuth client, powering the integration platform (e.g., Google Assistant/Home). Meanwhile, various *untrusted* third-party apps, including potentially malicious ones, take up the position of "trustworthy" AS in traditional OAuth. This reversal occurs because it is now the wide range of integrated apps that own resources and provide them to the platform by integrating with its open marketplace.

3 Threat Model

Generally speaking, there are three types of threats, corresponding to the three entities involved in OAuth:

- **Malicious App.** Untrusted third-parties can be taken up by malicious entities, as outlined in both paradigms in §2.3.
- **Malicious End-user.** The most prevalent form of OAuth threat, where attackers manipulate OAuth URLs at the user-agent and distribute to victim end-users. Some examples to be discussed in §4.1.
- **Malicious Platform.** An integration platform can theoretically be compromised, as studied in [13, 14]. If the resource or identity providers (*i.e.,* API or identity platforms in Fig. 3a) from the traditional OAuth paradigm turn malicious, attacks would also be feasible, as detailed in §6.

This paper assumes the threat model of *malicious apps*. We assume existing apps in the platform's marketplace are benign and secure. However, due to the OAuth role reversal (§2.3), new infiltration of malicious apps poses a significant risk.

Ease of Malicious App Infiltration.

- **Open Marketplace.** Any attacker can register an innocent-looking app with custom AS configuration and get listed in the open marketplace after the platform's vetting (a.k.a. certification) process.
- **Endure Vetting.** ASes are external servers that are black-box to the platform. Due to the possibility of server-side dynamic changes, the app's AS can keep malicious deeds dormant during vetting, only misbehave *after* approval.
- **Bypass Vetting.** In certain platforms, an attack can be launched without going through vetting, either by design (sharing-based app distribution, §2.2), or due to platform-side flaws (§4.3.1).

Attack Assumptions.

- **Attacker Capabilities.** Our attack aligns with the web attacker model [15], where an attacker is unable to alter the victim's network traffic, but capable of setting up victim-accessible malicious endpoints by integrating a malicious app.
- **Victim's Requirement.** The victim typically needs to be an end-user at the platform. They may have previously linked some of their app accounts to their platform account. The victim can be tricked into interacting with a malicious app on the platform or clicking an unassuming hyperlink.

Attack Scenario. Account linking is intended to involve only one app per OAuth flow (*i.e.,* the active app), as shown in Fig. 2. However, with the presence of a malicious app, two apps could get involved simultaneously, where an attacker-owned malicious app targets a benign app during the former's account linking process. The attacker's primary goal is to gain unauthorized access to or compromise the privacy of the victim's benign app account.

4 Platform-wide Cross-app OAuth Attacks

This section introduces the attacks and their extended impacts.

4.1 New Attack Surface in Integration Platform

Typical OAuth Attacks in Traditional Settings. In traditional OAuth deployment, two fundamental security requirements are `redirect_uri` matching by the AS and `state` matching by the OAuth client. Violations would result in well-known attacks under the *malicious end-user* threat model. Next, we illustrate them based on Fig. 2.

1) Regular Account Takeover. Before issuing an auth code in Step 4, the AS must match the pre-registered `redirect_uri` with the parameter value in the authorization request URL. Failure to do so allows an attacker to complete Step 1 and 2 as a normal end-user, but then distribute a manipulated authorization request URL (Step 3) with a `redirect_uri` pointing to their server. When the victim completes Step 3 and 4, the auth code would be leaked to the attacker's endpoint at Step 5. The attacker can then exchange the auth code for an access token with an authorization code injection attack [2, §4.5], leading to account takeovers.

2) Regular Login CSRF (Cross-Site Request Forgery). The `state` parameter should be bound to the user-agent or user session when issued at Step 2 (see [16, §5.3.5], [2, §4.7.1]), and returned unchanged in the authorization response (Step 4). This allows the OAuth client to match the returned `state` value against the initially issued value between Step 5 and 6. Failure to do so results in a login CSRF attack, where an attacker completes Step 1 to 4 and distributes Step 5's link to the victim, injecting the attacker's auth code into the victim's unsolicited OAuth flow, leading to forced authorization or login. The `state` parameter thus serves as a CSRF token.

Our Focus: Implications of Active App Tracking. While the security requirements above also apply to integration platforms, these measures are *insufficient* to secure a multi-app ecosystem. Similar impacts could be achieved through new attack vectors under the threat model of *malicious apps*.

Given the multiple integrated apps and their respective ASes, integration platform necessitates that its OAuth client *coherently track the active app* it interacts with during account linking, ensuring contact with the intended authorization and token endpoint. Since the access token request is not triggered until a callback, as a *functional requirement*, the platform shall embed the active app information in the OAuth flow, for later consumption at the redirection endpoint.

Reviewing Step 5 of Fig. 2, `state` and `code` are sent to the redirection endpoint specified by the `redirect_uri` parameter, typically with the platform session attached. The auth code itself does not indicate its originated app. Therefore, the platform's OAuth client commonly employs one of the two design choices: embedding the active app ID in the

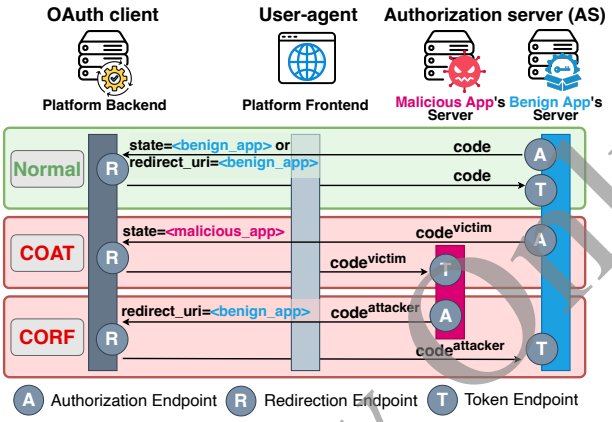

Figure 4: Insufficiency in Active App Tracking

OAuth `state` parameter (or within the platform's internal state, *e.g.,* a session, subsumed under `state` for conciseness), or embedding it in the `redirect_uri` parameter.

In normal operations, either design choice ensures that the authorization endpoint issuing the auth code (Fig. 2 Step 4) and the token endpoint redeeming it (Fig. 2 Step 6) belong to the same app, as shown in the "Normal" scenario of Fig. 4. However, this mechanism is insufficient (more precisely, flawed) to serve as a *security requirement*:

The `state` parameter only reflects how OAuth is started, but cannot track how it ends up. Since `state` is opaque to the ASes [1, §4.1.1], it may have traversed multiple ASes without the platform or the apps being aware. On the other hand, `redirect_uri` by itself has weak integrity, hence it can be tampered with by an AS. Consequently, as shown in the latter two scenarios in Fig. 4, the platform could be deceived by a malicious app, into either unknowingly leaking the auth code from a benign app to the malicious one (COAT attack, detailed in §4.2.1), or injecting an auth code from the malicious app into a benign one (CORF attack, §4.2.2).

4.2 Attack Details

In this subsection, we detail the two platform-wide cross-app attacks with the OAuth authorization code grant. The attacks are also applicable to the implicit grant [17], where the access token (instead of auth code) is directly issued and sent to the redirection endpoint via the user-agent, doing away with auth code exchange at the token endpoint. Table 1 presents a concise summary and comparison of Cross-app OAuth Account Takeover (COAT) with Request Forgery (CORF) attack.

4.2.1 Cross-app OAuth Account Takeover (COAT)

At the core of the attack is the malicious app's ability to 1) *redirect* from its authorization endpoint to that of a targeted benign app, while 2) still receiving the benign app's auth code at the malicious app's own token endpoint. This attack requires the OAuth client to track the active app using the

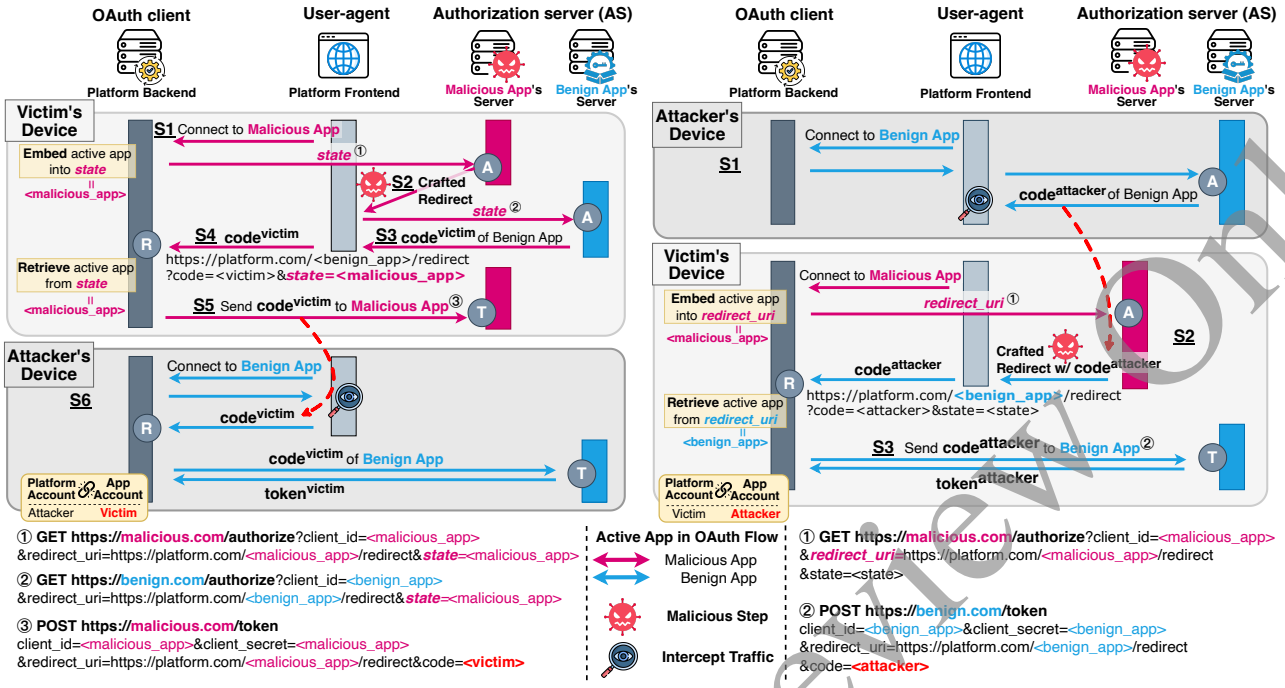

Figure 5: Attack Flow of Cross-app OAuth Account Takeover (COAT, Left) and Request Forgery (CORF, Right)

Table 1: A Comparison of COAT and CORF Attack

Aspects	COAT Cross-app OAuth Account Takeover	CORF Cross-app OAuth Request Forgery
Applicability	Distinguish active app by `state`	Distinguish active app by `redirect_uri`
Attack Setup	**Attacker:** Registers (must) and Publishes/Shares (optional) a Malicious app; **Victim:** Establishes Account Linking w/ Malicious app, potentially by a hyperlink click	
Execution Process	**Victim:** Malicious app *redirects* to Benign app's Authorization Endpoint, while retaining Malicious app's `state` ↪ Victim's Benign app auth code *leaked* to Attacker's Token Endpoint **Attacker:** Regular Auth Code Injection	**Attacker:** Prepares Attacker's Benign app auth code **Victim:** Malicious app changes active app in `redirect_uri` to Benign app and injects the auth code ↪ Attacker's Benign app auth code *injected* to Victim's OAuth flow
Consequence	**Victim**'s App Account linked w/ **Attacker**'s Platform Account	**Attacker**'s App Account linked w/ **Victim**'s Platform Account
Security Impact	Account Takeover (Unauthorized Access)	Forced Account Linking (Privacy Leakage)

`state` parameter. An end-to-end attack flow is depicted on the left-hand side of Fig. 5, with each step labeled as $\mathbf{S_n}$.

***Phase 1:* Attack Preparation.**

S0: The attacker registers an app in the platform, specifying their own authorization endpoint URL (*e.g.*, `https://malicious.com/authorize`) and token endpoint URL (*e.g.*, `https://malicious.com/token`). They shall be assigned an app ID and a `redirect_uri`. There are two variations of this setup, based on whether a distinct per-app `redirect_uri` (*e.g.*, `https://platform.com/<malicious_app>/redirect`, used as example in Fig. 5) or a universal URI for apps (*e.g.*, `https://platform.com/redirect`) is issued. We refer to the two variants as $\mathbf{COAT}_D$ and $\mathbf{COAT}_U$ respectively. Then, the attacker attempts to establish account linking with a targeted benign app and captures an actual authorization request URL to construct the attack logic of the malicious authorization endpoint that functions in Step 2.

***Phase 2:* Leaking Auth Code (Victim's Interaction).**

S1: The victim initiates account linking with the malicious app. The platform then directs the user-agent to the attacker-controlled authorization endpoint as specified in Step 0, with a freshly generated `state` parameter.

S2: Upon receiving the authorization request, the malicious app's authorization endpoint *redirects* to the benign app's authorization endpoint. The entire URL is replaced with the URL captured in Step 0, except for the `state` parameter, which is kept as the one just received (*i.e.,* the active `state` generated with victim's device).

S3: From the benign app AS's perspective, the authorization request appears no different from a genuine one directly initiated from the platform (*i.e.,* `redirect_uri`, `client_id`, and `scope`, if provided, are all correct). Given its prior linkage to the platform, the benign app issues an auth code without re-prompting end-user for explicit consent, as detailed in §4.3.2.

S4: The auth code, representing access to the victim's benign app account, is sent to the platform-hosted redirection endpoint (corresponding to the `redirect_uri`). An example URL is: `https://platform.com/<benign_app>/redirect?code=<victim>&state=<malicious_app>`. Following OAuth specification, the platform verifies that the `state` parameter is not tampered with, as it matches what was issued in Step 1 and is bound to the victim's user-agent.

S5: The account linking initiated in Step 1 was with the malicious app, the context of which embedded in or associated

with the `state` parameter. Consequently, the platform sends the auth code to the attacker-controlled token endpoint.

***Phase 3:* Attack Wrap-up.**
S6: *Authorization Code Injection.* The attacker then initiates a new OAuth flow with the benign app on their own platform console. Intercepting the network traffic, the attacker visits the redirection endpoint with a newly generated `state` parameter and the stolen auth code from the victim. The platform exchanges the code for an access token for the victim's benign app account, completing its linkage with the attacker's platform identity. The attacker can subsequently enjoy all the privileged access of the app, impersonating the victim.

4.2.2 Cross-app OAuth Request Forgery (CORF)

To defend against regular login CSRF, the `state` parameter should be mandated, tied to the platform's user session or user-agent, and matched against (see §4.1). However, this is inadequate for integration platforms, as a `state` generated for one app's account linking would also be valid for other apps. To accommodate active app tracking, some platforms in turn rely on the distinctive element in each app's pre-registered `redirect_uri`. However, this opens up the login CSRF attack under cross-app context, as shown on the right-hand side of Fig. 5, resulting in forced account linking.

***Phase 1:* Attack Preparation.**
S0: The attacker registers an app, specifying the authorization endpoint to a server under their control. They shall be assigned an app ID and a corresponding `redirect_uri` *e.g.*, `https://platform.com/<malicious_app>/redirect`. The attacker sets up the authorization endpoint for use in Step 2, intending to deceive the platform into believing it is interacting with a benign app's AS.
S1: The attacker initiates account linking with a benign app, authenticates at and authorizes the app with their own account. They intercept the authorization response to capture a valid auth code without redeeming it.

***Phase 2:* Victim's Interaction.**
S2: When the victim starts account linking with the malicious app, the attacker's malicious authorization endpoint returns a crafted authorization response that issues a redirection to `https://platform.com/<benign_app>/redirect?code=<attacker>&state=<state>`. This hits the redirection endpoint with the benign app's `redirect_uri`, along with the attacker's auth code obtained in Step 0 and the `state` parameter returned as its issued value.
S3: The platform backend receives the auth code from the attacker and perceives the active app as the targeted benign app based on the app ID in `redirect_uri`. Consequently, the platform contacts the benign app to redeem an access token and associates it with the victim's platform account. The token will also replace the app's existing token linked to the victim's platform account, if any.

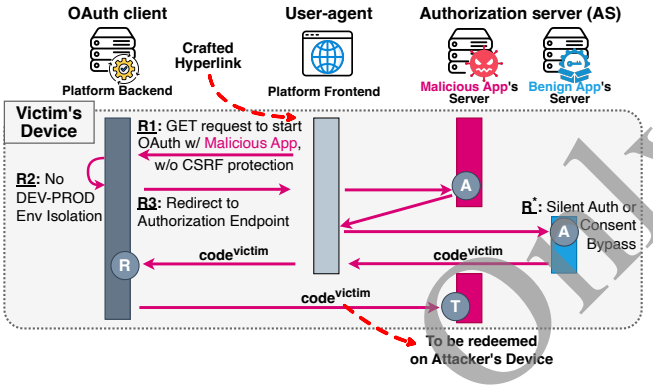

Figure 6: Worst-case Scenario: Single-click COAT Attack without Malicious App Distribution

Henceforth, whenever the victim interacts with the targeted benign app, all data will be saved to the attacker's benign app account, accessible remotely by the attacker. A detailed discussion of CORF attack's impact can be found in Appendix B.

4.3 Additional Impact

In several platforms, an attack *does not* necessitate the attacker to distribute or the victim to enable a malicious app. The victim also may not need to explicitly consent to the authorization with the targeted benign app. This can escalate the attack to *single-click*. We refer to this as the worst-case scenario, with a typical example illustrated in Fig. 6.

4.3.1 Platform-side Design Enabling Practical Attacks

In normal cases, the victim navigates the integration platform interface and enables a published malicious app to initiate the OAuth flow. If the following platform-side requirements (denoted as $\mathbf{R_n}$) are satisfied, the attack can be exempt from malicious app vetting (R2), and detached from platform-side interactions (R1 & R3):
R1: Starting OAuth with cross-site GET request. The mechanism to initiate an OAuth flow (as shown in Fig. 2 Step 1) is sending a GET request to the platform backend (*e.g.,* `https://platform.com/connect/<app_id>`), without any CSRF protection mechanisms in place. This enables practical phishing attacks even with the default `SameSite=Lax` cookie policy enforced in mainstream web browsers [18]. An attacker can simply share the request URL that initiates OAuth with the malicious app as a hyperlink, and tricking the victim to trigger the click.

Please note that single-click attacks by directly sharing an *authorization request URL* of the malicious app are infeasible, since it would break the binding of the `state` parameter in the URL with the user-agent or the platform's user session.
R2: Inadequate access control. The platform may lack isolation between development and production environments,

or only apply isolations on main functionalities of the apps (*e.g.*, prohibiting anyone from invoking a non-published app other than its developer), but misses the access control on OAuth. This allows a victim to establish account linking with a non-published app built by an attacker.

R3: Auto-triggered authorization request. The platform backend directly triggers the authorization request, typically with HTTP redirects, without further user interactions needed.

4.3.2 Circumventing Consent at Authorization Server

The above platform-side requirements are sufficient to conduct CORF in single-click. For COAT attacks, there is an additional requirement R* regarding the app's AS consent design.

R*: Adoption of silent authorizations. With an authenticated session and prior consent in place, many ASes are designed to *avoid* requesting explicit user consent again (*i.e.,* reprompting consent screen for previously-linked apps), but issue an auth code instantly. This design, commonly termed silent or automatic authorizations [19, 20], is adopted by big names such as Microsoft, GitHub, Dropbox and Xiaomi, which can be abused to make a COAT attack stealthier. Since all end-users on the same integration platform typically share the same pre-configured OAuth `client_id` per published app, an attacker-controlled malicious app in COAT can learn the targeted benign app's `client_id`, and then trigger silent authorizations if the victim has previously linked their benign app account to the platform.

Additionally, a COAT attacker has the potential to *bypass* explicit consent at ASes. Although several mainstream ASes support silent authorizations, they are disabled by default and only enabled for certain trusted authorization contexts due to security concerns. As a common design, a custom GET parameter (*e.g.*, `skip_confirm`, `force_reapprove`, `prompt`[6]) in the authorization request controls the consent screen's visibility. For instance, apps could register at trusted platforms with `https://app.com/authorize?prompt=none` for the authorization endpoint URL, and either set `prompt=consent` or omit the parameter at regular platforms. While this extra parameter can be tampered with at the user-agent during an OAuth flow, it poses no threat alone, due to the inability to directly share the authorization request URL for single-click attacks (as discussed in R1).

However, a COAT attacker can modify the parameter in the crafted redirection from the malicious app's authorization endpoint to that of the benign app (Step 2 of §4.2.1), bypassing the *consent* process. This technique, verified with GitHub, Dropbox and Xiaomi's AS, can also be applied to bypass *forced re-authentication* (observed in Dropbox) and *account selection* (seen in GitHub and Microsoft's AS).

Listing 1 illustrates an end-to-end single-click attack with consent bypass (derived from Yandex Smart Home's design).

[6]The `prompt` parameter is standardized in OIDC, but also widely adopted by ASes in OAuth for authorization.

```
[Benign Flow]
  POST https://platform.com/connect/<benign_app> w/ CSRF protection
  redirects to
  https://benign.com/authorize?prompt=consent&state=<benign_app>&...

[Malicious Flow (with consent bypass)]
  GET https://platform.com/connect/<malicious_app> // Send to victim
  redirects to
  https://malicious.com/authorize?state=<malicious_app>&...
  redirects to
  https://benign.com/authorize?prompt=none&state=<malicious_app>&...
  redirects to
  https://platform.com/redirect?code=<victim>&state=<malicious_app>
  POST (auto−sent by platform backend)
  https://malicious.com/token with code=<victim>&...
```

Listing 1: Single-click COAT Attack with Consent Bypass

5 Root Cause Analysis and Defense

In this section, we examine the root cause of cross-app OAuth vulnerabilities and explore a spectrum of countermeasures. We distinguish between solutions that address the issues from the root cause (§5.2.1), those that offer temporary relief (§5.2.2), and others that fall short (§5.2.3 and later in §6.3).

5.1 Root Cause

The essence of securing multi-app OAuth design is to avoid the platform's confusion in determining the active app throughout each OAuth flow. In a vulnerable case, the platform's OAuth client fails to ensure that the endpoint issuing the auth code belongs to the active app that redeems this code: Specifically, in COAT attacks, the platform merely relies on the `state` parameter to identify the malicious app as the active app. This is the app the platform initially contacts, but the platform is unaware that the authentic issuer of auth code is the benign app. Conversely, in CORF attacks, the platform relies on `redirect_uri` to identify the active app as the benign app, unaware that the auth code is actually issued by the platform's initial contact, which is the malicious app.

5.2 Defenses

The key to defending against cross-app vulnerabilities is verifying the consistency of the authorization context. Only the platforms have both the obligation and the capability to achieve this, as they alone, unlike individual apps, know which app is the platform's initial contact.

At the platform side, identifying the active app requires dual confirmation: 1) One indicating the app initially contacted by the platform, typically managed via the `state` parameter, where a unique identifier for each app is embedded. 2) Another confirmation should be one identifier that the app's AS can validate (or inherently trust) and subsequently return in the authorization response, to mark the actual issuer of the auth code. The platform's OAuth client must match the two

identifiers to ensure the active app is consistent, preventing the auth code from being sent to the wrong app's token endpoint.

Notably, the second identifier could potentially be app-specific (§5.2.1), or AS-specific (§6.3). The former, as the defense we suggest, turns out to be the more practical approach under the context of integration platforms.

5.2.1 Robust Countermeasure: App-specific Bindings

To universally defend all apps against COAT and CORF attacks, we propose to have a globally unique identifier generated for each app (*e.g.,* a random UUID-based app ID) during registration, which must be both associated with the `state` parameter and embedded into the `redirect_uri` path or subdomain. The `redirect_uri` is therefore a distinct app-specific URL, which can be validated and returned by the AS while preserving the original protocol flow. The platform MUST compare the app ID in `state` against its counterpart in `redirect_uri` at the redirection endpoint before proceeding with the access token request.

The above solution should be a *robust fix* for both COAT and CORF, but their migration steps vary:

Platforms affected by COAT$_U$. If the platform's original design is issuing universal `redirect_uri` for all apps (*i.e.*, COAT$_U$ as discussed in §4.2.1), the first step is to switch to distinct, app-specific `redirect_uris` for both existing and future apps. App developers must be informed to whitelist *only* this new `redirect_uri` at their AS. This migration requirement is considerably protracted since it involves coordinating with every third-party. A platform enforcing an immediate fix overnight can break those unmigrated apps, whose ASes will reject the new distinct URI from the platform. Conversely, it is impractical and insecure to wait until the migrations of all apps to be completed, which may take forever. Therefore, a transition period is likely needed to maintain backward compatibility, during which both distinct and universal `redirect_uris` shall be accepted by the platform.

Platforms affected by COAT$_D$. For these platforms, the defense is the most straightforward, requiring only that the platform updating their OAuth client backend code, extracting and matching the existing app ID in `state` and `redirect_uri`.

Platforms affected by CORF. Affected platforms shall update the format of the `state` parameter to embed or associate with the app ID (*e.g.,* as a new field in the JWT payload). Then, they should enforce matching against the distinctive element in `redirect_uri` at the redirection endpoint. The migration is expected to cause no compatibility issues, as `state` should be opaque to the AS.

5.2.2 Temporary Mitigations

Comparing the migration steps in mitigating different cross-app OAuth attacks, it is the most challenging to defend against COAT$_U$, which requires considerable communication and coordination overheads between the platform and all apps' developers. This section covers temporary measures to ease the transition before a complete fix is rolled out.

Securing apps which opted in for a fix. During the transition period, the platform may track individually whether an app has completed its migration and based on which include the old or new `redirect_uri` in the authorization request for that app. Apps which completed the migration (*e.g.,* through the developer console) will stick to *only* the new `redirect_uris`. To be specific, the platform must abort the OAuth flow if a universal `redirect_uri` is used in migrated apps, and enforce the app ID matching at OAuth client when a distinct `redirect_uri` is returned. The platform will remain as is for those unmigrated apps during the transition period. This is an effective and practical measure to let apps opt-in towards the complete fix and be secured as soon as possible.

Mitigating impacts on Single-clicks. Another track of mitigations is to eliminate single-click attack possibilities. One way is to ensure that every request to the platform in OAuth is CSRF protected (reflecting on R1 of §4.3.1), such that an OAuth flow cannot be inadvertently initiated or conducted by an end-user. In case this is hard to achieve, another approach, as introduced by some affected vendors, is to present a platform-enforced *extra consent screen* for unmigrated apps. As a temporary measure, it relies on end-users to hopefully reject malicious apps. Crucially, the platform-side consent screen must be presented *before* access token request (*e.g.,* at Step 1 or 5 of Fig. 2), otherwise it would be in vain as the auth code would have already been leaked.

5.2.3 Invalid Defense: PKCE

There is a common misconception that generic defenses like PKCE (Proof Key for Code Exchange) [21] is the cure to all attacks that involve an authorization code injection, and PKCE can always serve as an alternative CSRF protection. To clarify, although PKCE is supported by 3 platforms under our analysis, it does not mitigate COAT or CORF attacks.

In COAT, an attacker can force the victim to use a different PKCE code challenge generated within the attacker's OAuth flow, which can pass the validation at AS when attacker later exchanges the stolen auth code for a token. This is discussed by OAuth community under the name of "PKCE Chosen Challenge Attack" [22]. Conceptually similar, in CORF, the attacker may use the victim's code challenge to obtain an auth code, which is then injected back into the victim's OAuth flow. Fig. 7 demonstrates how CORF under PKCE can occur.

6 Comparison with IdP Mix-up Attack

Due to the OAuth "role reversal" paradigm shift in integration platforms (§2.3), cross-app attacks are lethally practical, as opposed to the IdP mix-up attacks discussed in prior literature.

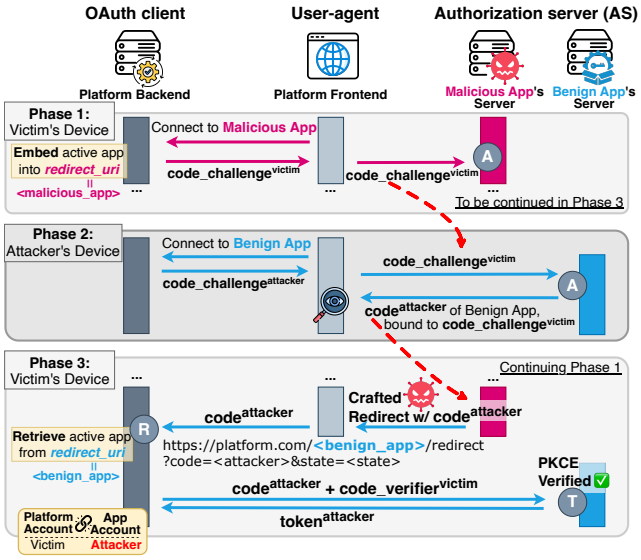

Figure 7: Attack Flow of CORF under PKCE Protection

Table 2: A Comparison of Cross-app *vs.* IdP Mix-up Attacks

Aspects	COAT / CORF	IdP Mix-up / Naïve RP Session Integrity [7, 23]
Ecosystem (§2.3)	Integration Platform	Ordinary Website
Mechanism (§2.3)	Account Linking (Authorization)	Authentication or Authorization
Attack Assumption (§6.2)	Malicious App (§3)	Malicious IdP (AS) (§6.1)
Typical Scenario (§6.2)	Open marketplace w/ Untrusted apps	Fixed set of Trusted IdPs
OAuth Registration (§6.2)	App's AS registers at Platform's OAuth client (§2.2)	RP (OAuth client) registers at IdP (AS) (§2.3)
Attack Practicality (§6.2)	Practical (§8)	Impractical
Defense (§6.3)	Per-app ID (§5.2)	Per-AS ID (`issuer`)

In this section, we reflect on trust relations and highlight the limitations of prior work in producing practical attacks and securing integration platforms.

6.1 IdP Mix-up Attacks in Traditional OAuth

Existing literature [7, 23] explores the possibilities of confusions among multiple IdPs (or ASes, more generally) in relation to the single RP of an application (*e.g.*, a website).[7] A victim end-user engaging with a malicious IdP could render leakage of their credentials, such as auth codes or access tokens, issued by an honest IdP (*IdP mix-up attack* in OAuth [7], *IdP confusion* in OIDC [23]); or alternatively, falsely accept an attacker's credentials at the honest IdP (*Naïve RP session integrity attack* [7]). For convenience, we collectively refer to them as *IdP mix-up attacks.*

Prior work focuses on identifying flaws at the protocol level, which ceased at conceptual attacks [7] or threat analysis in OAuth libraries [23]. The attacks were not introduced with the context of *multi-app* integration platform in mind, and lack end-to-end exploits identified in real-world deployments. Table 2 presents a comparison of cross-app attacks to existing discussions of IdP mix-up attacks in the OAuth community.

[7] Recall the mapping of AS *vs.* IdP and OAuth client *vs.* RP in §2.1.

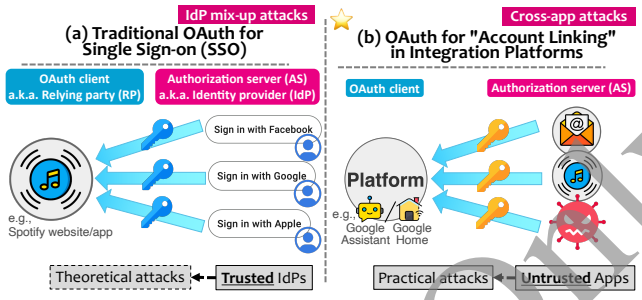

Figure 8: Impact of OAuth Role Reversal on Attack Practicality

6.2 Comparison of Trust Relationship

The distinction of practicality between the prior attacks in single-app multi-IdP scenario (*i.e.,* IdP mix-up attacks) and the attacks in integration platforms (*i.e.,* cross-app OAuth attacks) lies in the divergent trust relationships rooted in their architectures, as illustrated in Fig. 8.

(a) Difficulty of compromising a trusted party. In existing IdP mix-up attacks, the challenge lies in the involvement of a malicious IdP, be it *compromised* or *dynamically added* to the RP. This conflicts with the trusted nature between RP and IdPs. Take the multiple IdPs for website SSO as an example (Fig. 8a). Typically, each website supports only a *fixed* number of reputable IdPs, pre-registered by the RP. *Compromising* one of these existing, trusted IdPs like Google is thus highly unrealistic. In terms of *dynamically* establishing new RP-IdP relationships, OAuth AS metadata [24] and dynamic client registration [25] are available solutions but their functionalities are developer-oriented rather than exposed to end-users, which limit their practical use in attacks. When new IdPs are to be added, the registrations should be consciously initiated by the website's RP (developer) and therefore still imply a trusted relation.

(b) Ease of introducing an untrusted party. Contrastingly, under integration platform's context, well-known entities like Google rely on untrusted apps in an open marketplace to serve as the ASes (Fig. 8b). With the introduction of a malicious app, integration platform significantly eases the attack assumption of the infiltration of a malicious party. Our measurements in §8 find 16 out of 18 top-tier platforms vulnerable, with 9 susceptible to single-click attacks, impacting thousands of integrated apps internet-wide and millions of end-users.

6.3 Defense: per-app *vs.* per-AS ID

Based on academic work [7, 23], up-to-date IETF specifications [2, 3] propose an ad-hoc defense named `issuer` for IdP mix-up attack, which is a static *per-AS* identifier designated by the AS itself. Ideally, this defense does protect standards-compliant integration platforms and apps. However, in practice, the scope of these specifications does not consider the fundamental requirement in integration platforms

that *multiple apps can share the same* `issuer` (thus no longer unique), nor does it cover the setting of CORF. Additionally, deploying this defense in integration platforms is challenging, as most platforms and apps are not standards-compliant due to their unique requirements for manual OAuth registration, custom access token request logic, implicit grant support, etc.

As detailed in Appendix A, we believe that a *per-AS* ID (`issuer`) is a misaligned isolation boundary. Conversely, a platform-issued *per-app* ID, as we propose, is a more practical solution for securing OAuth in integration platforms.

7 Black-box Testing

Based on insights of the root cause, we derive a low-cost testing approach that enables scalable detection of cross-app OAuth vulnerabilities in real-world platforms.

7.1 Detection Logic

As established earlier, the key differentiator of cross-app vulnerabilities lies in how a platform's OAuth client identify the active app in an OAuth flow. However, the cloud-based, closed-source nature of most integration platforms obscures their OAuth implementation details, prompting the need for black-box or grey-box solutions for vulnerability detection.

Previous work has proposed detection approaches for IdP mix-up attacks. One approach is to operate as an insider, detecting vulnerabilities by registering as malicious and benign IdPs to simulate actual attacks [23]. However, in integration platforms, the development of an app is often labor-intensive due to the markedly different processes involved in each platform, where even skilled developers may find it challenging to learn. Another line of research [26, 27] operates as an outsider, identifying active attacks by real-time, browser-side monitoring of deviations from standard OAuth protocol flows. However, this method is ineffective and ill-suited for vulnerability detection, as protocol deviations is the prerequisite of but does not indicate a successful attack, and therefore cannot reflect the presence of vulnerabilities.

In summary, we envision a tool that 1) requires minimal manual efforts, eliminating the need for app registration or implementing ASes; and 2) can precisely detect vulnerabilities rather than merely identifying active attacks. To achieve this, we propose a lightweight, black-box testing approach that reveals the OAuth client's design in active app tracking. The key advantage is its ability to identify vulnerabilities with a small set of testing operations, and only rely on existing apps in the platform's marketplace for precise detection.

Key Insights. Our testing methodology is based on the following insights: 1) The OAuth client determines the active app at the platform's *backend* without explicitly disclosing the information to us. However, as MITM (man-in-the-middle) attackers, we can manipulate network traffic to induce app

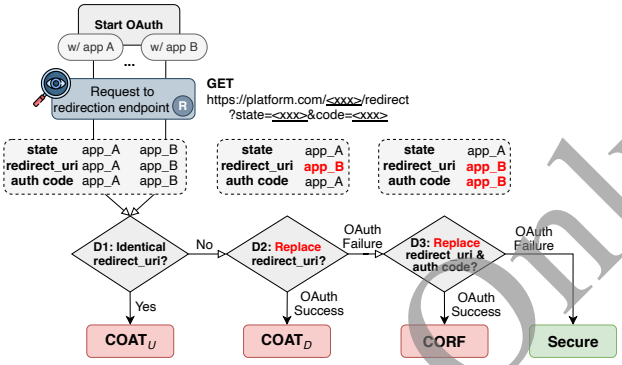

Figure 9: Decision Tree for Vulnerability Detection

identity inconsistencies to the platform's OAuth client. We can then inspect the platform's reaction in redeeming the auth code, so as to *infer* how the OAuth client tracks the active app. 2) Specifically, we aim to infer whether the platform *blocks* the access token request or *allows* it to proceed. If allowed, we need to further distinguish which app's AS the request is directed to. As a black-box tester, direct inspection of traffic on server-side is not possible. However, we can infer the destination by determining if an auth code can be accepted (*i.e.*, successfully redeemed) by the app's AS to which the code is routed. 3) The server-side results will manifest as an OAuth outcome (success or failure), which will be presented to the platform end-user and observable at the user-agent.

Based on these ideas, we present a decision tree-based approach as follows:

Initial Setup. In an integration platform, pick two apps (labeled as app A and app B) arbitrarily to establish account linking, note down their respective `redirect_uris`, as well as one unredeemed, valid auth code for each app.

Decision Phases. The core of our methodology is a decision tree consisting of 3 decisions (**D1**-**D3**), as depicted in Fig. 9 and explained in detail below:

D1. Does the platform assign identical `redirect_uri` for both apps?
- **Yes:** Vulnerable to **COAT**$_U$;
- **No:** Proceed to D2.

Explanations. D1 is based on the observation that a platform tracks the active app either by `state` or `redirect_uri`. Therefore, identical `redirect_uri` for both apps implies that the platform would *solely* depend on `state` to distinguish the active app, from which we can directly conclude the susceptibility of a COAT$_U$ attack (*i.e.*, an early exit). Otherwise, the platform may rely on `state`, `redirect_uri`, or both, to track the active app, which remain to be decided by D2 and D3.

D2. What is the OAuth outcome of replacing the distinctive element (*e.g.,* app ID) of `redirect_uri` in the request to the redirection endpoint?
- **Succeed:** Vulnerable to **COAT**$_D$;
- **Fail:** Proceed to D3.

Explanations. Based on the key insights, we intercept and modify the traffic to induce confusion between `state` and `redirect_uri` of the two apps. Specifically, we focus on the request to the redirection endpoint (Step 5 in Fig. 2), *e.g.,* GET to `https://platform.com/<app_A>/redirect?state=<app_A_state>&code=<app_A_code>`. In the OAuth flow with app A, we replace the identifier `app_A` with `app_B` in the URL path (or subdomain, depending on where the identifier is placed), *e.g.,* `https://platform.com/<app_B>/redirect?state=<app_A_state>&code=<app_A_code>`. If this identity mismatch can be discerned by the platform (*i.e.,* protection in place), an OAuth failure will be raised. Otherwise, two possibilities follows: if the platform sends the auth code to app A's token endpoint based on `app_A_state` (*i.e.*, COAT_D), then the OAuth flow will be a success; Conversely, if the platform directs the auth code to app B's token endpoint based on `app_B` in the URL path or subdomain (*i.e.*, CORF), since an unidentified auth code `app_A_code` is used, the OAuth flow will also fail.

To distinguish between the two possibilities that constitutes an OAuth failure in D2, a further decision D3 is introduced:

D3. On top of the substitution made in D2, what is the OAuth outcome of replacing the auth code as well?
- **Succeed:** Vulnerable to **CORF**;
- **Fail:** Protection in place, secure.

Explanations. If we replace auth code in addition to app ID in the `redirect_uri`, such as `https://platform.com/<app_B>/redirect?state=<app_A_state>&code=<app_B_code>`, the OAuth flow will still fail if proper check is in place, but a platform vulnerable to CORF can be identified if the OAuth outcome is a success.

By this means, we can distinctly identify the presence of a defense, and, in a vulnerable case, discern the active app's tracking mechanism. The branching in D2 and D3 derives reliable conclusions by addressing ambiguities regarding whether an OAuth failure stems from defensive measures or misrouted auth code redemptions.

7.2 Vulnerability Detection Pipeline

To measure the prevalence and severity of cross-app OAuth vulnerabilities in real-world integration platforms, we conduct testing following a three-phase pipeline. First, we select a curated list of integration platforms. Next, we utilize COVScan, a black-box scanner based on the decision tree in §7.1 for vulnerability detection. Finally, we perform manual verification for PoC exploitation.

7.2.1 Platforms Collection

Our approach for collecting the set of integration platforms potentially susceptible to the cross-app OAuth attacks is as follows: We document the supported platform list of several cross-platform apps (*e.g.,* TickTick [28], a to-do list app). In parallel, we survey the representatives in each category of integration platforms. The combined list of the above two steps is then filtered to include only those with an *open* marketplace design, ensuring the feasibility for a malicious app to engage.

7.2.2 Semi-automated Vulnerability Detection

The detection logic in §7.1 is handy for manual testing with a web debugging proxy. Building on this, we further develop COVScan, a low-cost testing tool to facilitate scalable detection of Cross-app OAuth Vulnerabilities.

Tool Design. COVScan tracks the OAuth flow using a state machine and modifies intercepted traffic to assess the (in)security of each platform. The state machine is modeled to capture each OAuth step within the traffic and only transition to the next state when the request and response comply with the OAuth standard (*i.e.,* correct sequence and syntax). The detection process begins with app B by gathering its `redirect_uri` and a valid auth code without redeeming it. Then, COVScan instructs the end-user to initiate a new flow with app A, also collecting the two, and vets against D1. If the `redirect_uri` is distinct, the tool releases the modified request as per D2. If no decisive conclusion is reached, the process repeats once in a *new* OAuth flow with app A, this time releasing the modified request following D3. Based on the judgment of D1 and the OAuth outcomes of D2 and D3, a reliable conclusion is derived.

A crucial task is distinguishing between successful and failed OAuth flows. COVScan currently monitors up to three HTTP requests and responses at the user-agent, starting from the request to the redirection endpoint. It filters request URLs and response bodies using a keyword list that signifies a failed OAuth outcome (*e.g.*, "error", "unsuccessful", "invalid"). The tool is semi-automated because it requires minimum human intervention navigating the platform, which is inevitable due to UI variations across platforms and the authentication needs of the apps, yet the tool remains user-friendly for laypersons.

Implementation. We implement COVScan in Python, which utilizes `selenium-wire` [29] to accomplish human interaction-assisted browser automation while being aware of the underlying HTTP traffic. The tool also adopts `undetected-chromedriver` [30] to bypass potential bot detection (*e.g.,* human verification by cloudflare). For mobile-based platforms where Selenium is incompatible, we develop an alternative version based on `mitmproxy` [31].

Discussions. To induce inconsistencies in app identifiers, we opt to replace the distinct element in `redirect_uri` rather than substituting the `state` parameter. The rationale behind is in several platforms' designs, at most one `state` is considered valid for each platform end-user. Thus, substituting the `state` will always fail an OAuth flow, introducing false negatives in identifying COAT_D and CORF. Furthermore, as

mentioned earlier, D2 and D3 should be conducted in two *separate* OAuth flows with the same app. We argue that this is the minimum setup; otherwise, D3 will be reusing the `state` parameter of D2, which contravenes the one-time use property of `state` and leads to false negatives in identifying CORF.

Limitations. One potential limitation of COVScan is the occurrence of false positives and negatives due to the adoption of OAuth failure keyword list, the mechanism to distinguish OAuth outcomes. False negatives may arise if the list is too general, as the keywords may be hit by normal HTTP traffic. Conversely, a list too specific might lead to false positives by falsely flagging a secure platform as vulnerable to COAT_D or CORF attack. To address this issue, we refine the keyword list based on empirical observations and could further incorporate LLM to facilitate understanding OAuth outcomes.

7.2.3 Attack Verification and Impact Evaluation

For platforms identified as vulnerable by COVScan, we follow §4.2 to conduct PoC attacks, and evaluate metrics from §4.3 to identify potential amplifications in security impact. The PoCs serve as the ground truth of our semi-automated vulnerability detection and are used as proof for responsible disclosure.

Note that for platforms allowing unvetted, sharing-based app distribution, we replicate the attack flow using two platform accounts to carry out the develop-distribute-interact process with the malicious app. For platforms that require vetting before app distribution, we conduct testing using the same platform account as the malicious app developer, refraining from publishing the app due to ethical concerns.

8 Evaluation

Based on the detection pipeline in §7.2, we collected 24 mainstream integration platforms and filtered out 6 platforms without their marketplace open to third-party developers (*e.g.*, Integrately [32]). With the help of COVScan, we systematically evaluated the (in)security of the remaining 18 integration platforms regarding their cross-app OAuth vulnerabilities.

8.1 Empirical Results

The findings of our measurement study are presented in Table 3 and 4, accompanied by the following key observations:

Prevalence of Vulnerabilities. Out of 18 platforms analyzed, 16 are susceptible to cross-app OAuth attacks, including 4 out of 6 workflow automation platforms, all 8 virtual assistants and all 4 smart homes. Specifically, 11 platforms are vulnerable to COAT attacks: 7 to the COAT_U variant, 5 to the COAT_D variant, including 1 platform susceptible to both. Additionally, 5 platforms are subject to CORF attacks. Furthermore, 9 platforms fall in the worst-case scenario, where an end-user can be compromised with a single click and without the need of the attacker to distribute the malicious app.

Defenses Deployed. While neither standardized nor publicly discussed, the two secure platforms (Zapier and IFTTT) adopt solutions similar to §5.2.1, which associate an app-specific ID with `state`, embed an equivalent ID in `redirect_uri`, and enforce matching of the two at the redirection endpoint. Moreover, their design for the distinct element in `redirect_uri` uses a globally unique app name for published apps, and also a globally unique app name or numeric identifier for custom apps. This naming convention effectively prevents naming conflicts and potential `redirect_uri` collisions.

Feasibility of Semi-automated Detection. COVScan can reliably detect vulnerabilities or confirm the security in all platforms, producing consistent results as manual exploitation which serves as the ground truth. Note that 3 of the vulnerable platforms are labeled as N/A in Table 3 because they fixed in early stage of our research, prior to the development of COVScan. Despite this, we later applied the tool to these platforms post-fix and successfully confirmed their security.

Evaluation on Open-source Implementations. Aside from the 18 closed-source platforms listed in Table 3, we further investigate 4 trending open-source workflow automation platforms on GitHub, with regard to their `redirect_uri` setup and the presence of defense against cross-app OAuth attacks. The evaluation result is presented in Table 4.

This evaluation serves two purposes: 1) Protecting against downstream applications. The vulnerable platforms require immediate action, as downstream applications that have utilized these open-source platforms for OAuth are also affected. For instance, Trigger.dev [33], a platform incorporating Nango for account linking, also issues universal `redirect_uris` and is therefore likewise vulnerable to COAT_U. 2) Validating testing approaches. Regarding the presence of defense, we rely on implementation within the open-source codebase as the fundamental ground truth to further validate the effectiveness of COVScan (§7.2.2) and the validity of our manual PoC exploitation (§7.2.3). No inconsistencies have been observed.

8.2 Case Studies

Microsoft Power Automate. A critical COAT vulnerability [4] was identified in Microsoft Power Automate [34], a workflow automation platform for both consumer and enterprise use. This vulnerability enables platform-wide, single-click account takeovers, compromising over 50 *first-party* Microsoft applications in the Microsoft 365 suite (*e.g.,* Outlook, OneDrive) and Azure infrastructure (*e.g.,* Key Vaults, SQL Servers). For example, an attacker can gain full access to a victim's Outlook emails or read their Azure Key Vault secrets without explicit consent.

The single-click exploit builds upon the following insights: 1) While the platform supports publishing and sharing-based

Table 3: Findings of Platform-wide Cross-app OAuth Attacks Among Mainstream Integration Platforms

Type	Platform	ℂ	𝔼	# Users	COAT		CORF	Detectable	Attack Vector	
					$COAT_U$	$COAT_D$			App Distribution	Single-Click
Workflow Automation Platforms 𝔼,ℂ	Microsoft Power Automate	✓	✓	33M MAU	💀	💀		✓	*Share*, *Publish*	✓
	IFTTT	✓		27M				✓	N/A	N/A
	Zapier	✓	✓	2.2M				✓	N/A	N/A
	A Business Collab. Platform	✓	✓	54M MAU	💀			✓	*Share*	✓
	Workato		✓	21K Orgs	💀			✓	*Share*, *Publish*	
	A Top-tier iPaaS		✓	70K Companies	💀			✓	*Publish* + *Share*	✓
Virtual Assistants 𝔼,ℂ	A leading LLM platform	✓	✓	180M WAU			💀	✓	*Share*, *Publish*	
	ByteDance Coze	✓	✓	2M MAU	💀			✓	*Share*, *Publish*	✓
	Google Assistant	✓		500M MAU		💀		N/A [1]	*Share*, *Publish*	
	Amazon Alexa	✓		100M		💀		✓	*Share*, *Publish*	✓
	Samsung Bixby	✓		200M		💀		N/A	*Publish*	
	Xiaomi XiaoAI	✓		115M			💀	✓	*Publish*	✓
	Baidu Xiaodu	✓		40M			💀	✓	*Publish*	✓
	Alibaba AliGenie	✓		40M			💀	✓	*Publish*	
Smart Homes ℂ	Google Home	✓		500M Installs		💀		N/A	*Share*, *Publish*	
	Samsung SmartThings	✓		285M	💀			✓	*Share*, *Publish*	✓
	Xiaomi Mi Home	✓		83M			💀	✓	*Publish*	
	Yandex Smart Home	✓		45M	💀			✓	*Share*, *Publish*	✓
Total	18	16	7		7	5	5	15		9

ℂ: Consumer-facing platform; 💀: Vulnerable platform. [1] Vulnerability fixed before COVScan came into being.
𝔼: Enterprise-facing platform, also referred to as iPaaS (Integration Platform as a Service) by workflow automation platforms.
$COAT_U$: COAT with universal `redirect_uri` for multiple apps; **$COAT_D$**: COAT with distinct (per-app) `redirect_uris`.
Detectable: Platform's (in)security detectable by COVScan. **Single-Click**: Feasibility of Single-click attack w/o malicious app distribution.
Share: Share a custom app individually w/o vetting; *Publish*: Go through vetting process and publish in marketplace.

Table 4: Evaluation of Open-source Workflow Automation Platforms

Platform	# Stars	`redirect_uri`	COAT/CORF Defense
n8n	50.1K	Universal	Missing, **$COAT_U$**
Activepieces	10.1K	Universal	Missing, **$COAT_U$**
Automatisch	6.4K	Distinct	Missing, **$COAT_D$**
Nango	4.8K	Universal	Missing, **$COAT_U$**

app distribution, even stealthier, an attacker can directly distribute the hyperlink that initiates OAuth with their non-distributed malicious app. The link will still work in the victim's user-agent, redirecting to the app's authorization endpoint (R3 in §4.3.1, same below) given no CSRF protection (R1) and isolation between environments (R2) in place. 2) The platform not only hosts third-party apps, but also Microsoft first-party services. First-parties are implicitly trusted by design, where authorization consent is *never* needed (even more severe than R^*). 3) For first-party apps, the only required user interaction—a confirmation page to select the Microsoft account being used in account linking—can be circumvented following the strategy in §4.3.2. 4) The platform issues distinct `redirect_uris` for first-party apps and universal ones for third-parties, tracking the active app solely by the `state` parameter. This enables $COAT_D$ and $COAT_U$ attacks.

Alarmingly, a victim does not need to be using or even registered at the platform. Merely having a non-expired Microsoft account session at the user-agent suffices to get compromised in single click. This puts any Microsoft user with a 365 or Azure subscription at risk. In addition to first-parties, the broader ecosystem, namely all OAuth-enabled *third-party* apps (*e.g.,* GitHub and Dropbox) are also impacted.

In response, Microsoft "ripped apart, rearchitected the whole connectors ecosystem" [35]. They also deprecated the universal `redirect_uri` and instructed app developers to migrate to distinct ones in order to enforce the robust fix (§5.2.1), while putting up an extra consent screen during the transition period (§5.2.2).

A Leading LLM Platform. An emerging LLM-empowered assistant platform tracks the active app (*i.e.,* plugin) via distinct `redirect_uris`, without even issuing a `state` parameter. This omission renders it susceptible to regular login CSRF in the first place. Following our initial communication, their engineering team applied a fix to generate and mandate the `state` parameter. However, the platform remained exposed to CORF, which was ultimately addressed after another round of contact.

9 Related Work

Security of Integration Platforms. Existing research investigates the unique security challenges in each type of integration platform (*e.g.,* exploiting automation rules [36], voice squatting [37], LLM prompt injections [38] and flawed smart home control protocols [39]). Notably, there is a consistent line of research on the threat model of malicious or compromised platforms [13, 14] that re-architect the platforms to achieve certain security guarantees. In this work, we adopt the threat model of malicious apps, while assuming a trusted platform. We inspect intrinsic issues in the use of OAuth protocol across different types of integration platforms to fulfill

account linking.

OAuth Security. The security of OAuth protocol [1] has undergone scrutiny through various aspects, including formal analysis [7], penetration testing [40], and (semi-)automated evaluations [41–43]. Nevertheless, most existing literature and industrial efforts focus on examining ordinary websites, OAuth libraries, or emerging use cases like mobile [11,44] and dual-window SSOs [45]. Yet, integration platforms, which introduce a larger-scale OAuth architecture with more complex trust relationships, have been largely overlooked.

IdP Mix-up Attack. Prior work [7, 23, 46] discusses the feasibility of OAuth attacks when a victim is tricked into interacting with a malicious IdP, if there are multiple IdPs configured for an RP. However, these studies lack discoveries of concrete exploits in real-world deployments, since the assumptions under their contrived threat model are too demanding (*e.g.,* compromising an IdP). In contrast, as open ecosystems, the inherent risk of hosting apps in integration platforms relaxes the attack assumptions. While the OAuth community further standardized mix-up attack mitigation in [2, 3], the suggested defenses are not tailored for integration platforms.

Towards identifying IdP mix-up attacks, PrOfESSOS [23] detects vulnerabilities by registering as malicious and benign IdPs and simulating an actual attack. WPSE [26] and Bulwark [27] identify active attacks through real-time monitoring with browser add-ons, blocking deviations from the intended protocol flow. By contrast, COVScan conducts decision tree-driven black-box testing in integration platforms as an external entity. Our approach requires no knowledge of platform internals or app development and proves effective in identifying cross-app OAuth vulnerabilities.

10 Conclusion

Integration platforms offer an all-in-one solution that seamlessly connects various apps for aggregated control, streamlining our digital lives. In this paper, we dissect the security threat posed by untrusted apps in OAuth-based account linking within integration platforms, revealing the first cross-app OAuth attacks with real-world exploits. We design a semi-automated vulnerability scanner to facilitate scalable detection. Our study shows that 16 out of 18 integration platforms are susceptible to these attacks, with the worst cases enabling single-click account takeovers. The prevalence and severity motivate our robust defense to fully mitigate the vulnerabilities. Our research underscores the importance of re-inspecting OAuth under the multi-app architecture and role reversal paradigm of integration platforms.

Ethical Concerns and Responsible Disclosure

All testing on live systems is conducted with our test accounts. The malicious apps in PoCs are all test apps we developed ourselves and we never attempted to publish the apps in marketplace. For decision tree-based testing, although real-world apps are involved, we never attempted to penetrate or send large amounts of traffic to their services. Moreover, all errors produced by our testing methodology are already covered by standard error handling in OAuth specifications.

As of November 2024, we have informed all 16 vulnerable closed-source platforms, and more than 120 days have passed since our reports. Of these, 11 platforms have responded, with 9 having already patched the vulnerabilities and 2 still working on it. Among the fixed platforms, 6 applied the robust fix (§5.2.1), while 3 implemented an extra consent screen (§5.2.2). We are also working with developers of open-source platforms to help them understand the implications.

This research was initiated at Samsung Bixby, where the attack vector was internally identified and fixed at early stage. Following our bug reports, we have received $35K bug bounty rewards as well as acknowledgments from multiple vendors. For example, Google acknowledged and fixed their issue within two weeks upon our report; Microsoft invited us to a collaboration meeting and issued CVE-2023-36019 [4], a critical-severity CVE with a 9.6 CVSS score.

We have contacted the IETF OAuth Working Group to seek incorporation of our findings as normative changes into specifications such as the OAuth 2.0 Security Best Current Practice [2] or the OAuth 2.1 [47] draft. We will also present our research at the OAuth Security Workshop [48] to communicate with the broader OAuth community.

Acknowledgments

We sincerely thank the reviewers for their valuable feedback. In particular, we thank our shepherd for the constructive suggestions and guidance throughout the revision process, which helped enhance the paper considerably. This research is supported in part by the CUHK MobiTeC Fund (project# 6901539) and the CUHK Strategic Impact Enhancement Fund (project# 399857576).

References

[1] Dick Hardt. The OAuth 2.0 Authorization Framework. RFC 6749, October 2012.

[2] Torsten Lodderstedt, John Bradley, Andrey Labunets, and Daniel Fett. OAuth 2.0 Security Best Current Practice. Internet-Draft draft-ietf-oauth-security-topics-29, Internet Engineering Task Force, June 2024. Work in Progress.

[3] Karsten Meyer zu Selhausen and Daniel Fett. OAuth 2.0 Authorization Server Issuer Identification. RFC 9207, March 2022.

[4] Security Update Guide - Microsoft. CVE-2023-36019 - Microsoft Power Platform Connector Spoofing Vulnerability. https://msrc.microsoft.com/update-guide/vulnerability/CVE-2023-36019.

[5] Hui Wang, Yuanyuan Zhang, Juanru Li, and Dawu Gu. The achilles heel of oauth: a multi-platform study of oauth-based authentication. In *Proceedings of the 32nd Annual Conference on Computer Security Applications*, pages 167–176, 2016.

[6] Louis Jannett, Christian Mainka, Maximilian Westers, Andreas Mayer, Tobias Wich, and Vladislav Mladenov. Sok: Sso-monitor-the current state and future research directions in single sign-on security measurements. In *2024 IEEE 9th European Symposium on Security and Privacy (EuroS&P)*, pages 173–192. IEEE, 2024.

[7] Daniel Fett, Ralf Küsters, and Guido Schmitz. A comprehensive formal security analysis of oauth 2.0. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1204–1215, 2016.

[8] Nat Sakimura, John Bradley, Mike Jones, Breno De Medeiros, and Chuck Mortimore. Openid connect core 1.0 incorporating errata set 2. *OpenID Connect WG, Specification*, 2023.

[9] Michael B. Jones, John Bradley, and Nat Sakimura. JSON Web Token (JWT). RFC 7519, May 2015.

[10] John Bradley, Torsten Lodderstedt, and Hans Zandbelt. Encoding claims in the OAuth 2 state parameter using a JWT. Internet-Draft draft-bradley-oauth-jwt-encoded-state-09, Internet Engineering Task Force, November 2018. Work in Progress.

[11] William Denniss and John Bradley. OAuth 2.0 for Native Apps. RFC 8252, October 2017.

[12] Microsoft Learn. Investigate and remediate risky oauth apps - microsoft defender for cloud apps. https://learn.microsoft.com/en-us/defender-cloud-apps/investigate-risky-oauth.

[13] Earlence Fernandes, Amir Rahmati, Jaeyeon Jung, and Atul Prakash. Decentralized action integrity for trigger-action iot platforms. In *Proceedings 2018 Network and Distributed System Security Symposium*, 2018.

[14] Yunang Chen, Amrita Roy Chowdhury, Ruizhe Wang, Andrei Sabelfeld, Rahul Chatterjee, and Earlence Fernandes. Data privacy in trigger-action systems. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 501–518. IEEE, 2021.

[15] Devdatta Akhawe, Adam Barth, Peifung E Lam, John Mitchell, and Dawn Song. Towards a formal foundation of web security. In *2010 23rd IEEE Computer Security Foundations Symposium*, pages 290–304. IEEE, 2010.

[16] Torsten Lodderstedt, Mark McGloin, and Phil Hunt. OAuth 2.0 Threat Model and Security Considerations. RFC 6819, January 2013.

[17] OAuth Community Site. OAuth 2.0 Implicit Grant Type. https://oauth.net/2/grant-types/implicit/.

[18] MDN Web Docs. Set-cookie - http | mdn. https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Set-Cookie#browser_compatibility.

[19] San-Tsai Sun and Konstantin Beznosov. The devil is in the (implementation) details: an empirical analysis of OAuth SSO systems. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 378–390, 2012.

[20] Auth0 Docs. Configure silent authentication. https://auth0.com/docs/authenticate/login/configure-silent-authentication.

[21] Nat Sakimura, John Bradley, and Naveen Agarwal. Proof Key for Code Exchange by OAuth Public Clients. RFC 7636, September 2015.

[22] IETF 105. PKCE chosen challenge attack. https://datatracker.ietf.org/meeting/105/materials/slides-105-oauth-sessa-oauth-security-topics-00.

[23] Christian Mainka, Vladislav Mladenov, Jörg Schwenk, and Tobias Wich. Sok: single sign-on security—an evaluation of openid connect. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 251–266. IEEE, 2017.

[24] Michael B. Jones, Nat Sakimura, and John Bradley. OAuth 2.0 Authorization Server Metadata. RFC 8414, June 2018.

[25] Justin Richer, Michael B. Jones, John Bradley, Maciej Machulak, and Phil Hunt. OAuth 2.0 Dynamic Client Registration Protocol. RFC 7591, July 2015.

[26] Stefano Calzavara, Riccardo Focardi, Matteo Maffei, Clara Schneidewind, Marco Squarcina, and Mauro Tempesta. WPSE: Fortifying web protocols via Browser-Side security monitoring. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 1493–1510, Baltimore, MD, August 2018. USENIX Association.

[27] Lorenzo Veronese, Stefano Calzavara, and Luca Compagna. Bulwark: Holistic and verified security monitoring of web protocols. In *Computer Security–ESORICS 2020: 25th European Symposium on Research in Computer Security, ESORICS 2020, Guildford, UK, September 14–18, 2020, Proceedings, Part I 25*, pages 23–41. Springer, 2020.

[28] Integrations - ticktick. https://ticktick.com/integrations.

[29] wkeeling. selenium-wire: Extends selenium's python bindings to give you the ability to inspect requests made by the browser. https://github.com/wkeeling/selenium-wire.

[30] ultrafunkamsterdam. undetected-chromedriver: Custom selenium chromedriver. https://github.com/ultrafunkamsterdam/undetected-chromedriver.

[31] Aldo Cortesi, Maximilian Hils, Thomas Kriechbaumer, and contributors. mitmproxy: A free and open source interactive HTTPS proxy, 2010–. https://mitmproxy.org/.

[32] 20 million+ ready automations for 1100+ apps | integrately. https://integrately.com/.

[33] Trigger.dev | the open source background jobs framework. https://trigger.dev/.

[34] Microsoft. Microsoft power automate – process automation platform. https://www.microsoft.com/en-us/power-platform/products/power-automate.

[35] Scott Gorlick. Q&A Session of "Security Research in Copilot Studio". https://youtu.be/0Bw2YCDypUY?t=2743.

[36] Qi Wang, Pubali Datta, Wei Yang, Si Liu, Adam Bates, and Carl A Gunter. Charting the attack surface of trigger-action iot platforms. In *Proceedings of the 2019 ACM SIGSAC conference on computer and communications security*, pages 1439–1453, 2019.

[37] Nan Zhang, Xianghang Mi, Xuan Feng, XiaoFeng Wang, Yuan Tian, and Feng 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)*, pages 1381–1396. IEEE, 2019.

[38] OWASP Foundation. Owasp top 10 for large language model applications. https://owasp.org/www-project-top-10-for-large-language-model-applications/.

[39] Wei Zhou, Yan Jia, Yao Yao, Lipeng Zhu, Le Guan, Yuhang Mao, Peng Liu, and Yuqing Zhang. Discovering and understanding the security hazards in the interactions between IoT devices, mobile apps, and clouds on smart home platforms. In *28th USENIX security symposium (USENIX security 19)*, pages 1133–1150, 2019.

[40] PortSwigger. Oauth 2.0 authentication vulnerabilities | web security academy. https://portswigger.net/web-security/oauth.

[41] Ronghai Yang, Wing Cheong Lau, Jiongyi Chen, and Kehuan Zhang. Vetting single Sign-On SDK implementations via symbolic reasoning. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 1459–1474, Baltimore, MD, August 2018. USENIX Association.

[42] Pieter Philippaerts, Davy Preuveneers, and Wouter Joosen. Oauch: Exploring security compliance in the oauth 2.0 ecosystem. In *Proceedings of the 25th International Symposium on Research in Attacks, Intrusions and Defenses*, pages 460–481, 2022.

[43] Tamjid Al Rahat, Yu Feng, and Yuan Tian. Cerberus: Query-driven scalable vulnerability detection in oauth service provider implementations. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 2459–2473, 2022.

[44] Eric Y. Chen, Yutong Pei, Shuo Chen, Yuan Tian, Robert Kotcher, and Patrick Tague. Oauth demystified for mobile application developers. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 892–903, 2014.

[45] Louis Jannett, Vladislav Mladenov, Christian Mainka, and Jörg Schwenk. Distinct: Identity theft using in-browser communications in dual-window single sign-on. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 1553–1567, 2022.

[46] Daniel Fett. Mix-up, revisited - danielfett.de. https://danielfett.de/2020/05/04/mix-up-revisited/.

[47] Dick Hardt, Aaron Parecki, and Torsten Lodderstedt. The OAuth 2.1 Authorization Framework. Internet-Draft draft-ietf-oauth-v2-1-12, Internet Engineering Task Force, November 2024. Work in Progress.

[48] OAuth Security Workshop - OSW 2025. https://oauth.secworkshop.events/osw2025.

[49] Chetan Bansal, Karthikeyan Bhargavan, Antoine Delignat-Lavaud, and Sergio Maffeis. Discovering concrete attacks on website authorization by formal analysis. *Journal of Computer Security*, 22(4):601–657, 2014.

[50] Alexa Skills Kit. Configure an implicit grant. https://developer.amazon.com/en-US/docs/alexa/account-linking/configure-implicit-grant.html.

[51] Google for Developers. Account linking with oauth | actions on google account linking. https://developers.google.com/assistant/identity/oauth2?oauth=implicit#flow.

[52] Zapier Docs. Use code mode to refine your api call. https://platform.zapier.com/build/code-mode.

[53] Workato Docs. How-to guides - authentication - authorization code grant. https://docs.workato.com/developing-connectors/sdk/guides/authentication/oauth/auth-code.html#auth-code-grant-variations.

[54] Bixby Developer Center. token-endpoint. https://bixbydevelopers.com/dev/docs/reference/type/authorization.user.oauth2-custom.token-endpoint.

A Limitations of `issuer` Defense for Integration Platform

The defense against IdP mix-up attack requires each AS to return a unique ID that identifies itself (the OAuth AS metadata-dependent `issuer` or its fallback option). However, this per-AS isolation boundary is not well-suited for multi-app integration settings for the following reasons:

Misaligned Preconditions.

1. **The "duplicate `issuer`" problem:** According to IETF specifications [2, §4.4.1], the OAuth client should track "the *authorization server* chosen by the user" as a precondition to fall within the scope of IdP mix-up attacks. However, this conflicts with the requirement of integration platforms, where multiple apps can legitimately share the same `issuer` (*e.g.,* an official Dropbox app and a custom Dropbox app with different OAuth scopes, both using the same AS). As a result, an integration platform MUST track by a *per-app* ID as a functional requirement (§4.1). On the other hand, tracking by a *per-AS* ID (`issuer`) is an inappropriate isolation boundary, as it is no longer a unique identifier. Platforms cannot determine which app's AS to send the auth code to if multiple apps share the same `issuer`. Consequently, platforms blindly following the specification would be forced to break or expel existing `issuer`-sharing apps.
2. **Lack of CORF coverage:** The precondition, attack and targeted defense of CORF (as well as the corresponding Naïve RP session integrity attack [7] in traditional OAuth scenarios) are never covered by any IETF specifications. For example, the specification [2, §4.4.1] requires OAuth clients to track by "a session bound to the user's browser" (or by session-bound `state` as we put it) as the mix-up attack precondition. However, as a functional requirement, CORF-affected platforms track by distinct `redirect_uris` instead (§4.2.2).[8] Therefore,

[8]The specification does mention that tracking by "Per-AS Redirect

platforms that initially adopt the latter design (*e.g.,* the leading LLM vendor discussed in §8.2 as a case study) would not be aware that they are affected, because they do not meet the specified preconditions.

Impractical Defense.

1. **Lack of OAuth AS metadata support:** Typically, the `issuer` value is sourced from the OAuth AS metadata [24] to ensure its authenticity (*i.e.*, ensuring that `issuer` cannot be forged by verifying domain ownership). However, in the platforms we surveyed, *none* support metadata, and this feature is also rarely supported by existing integrated apps. Instead, all platforms require app developers to manually register their ASes (as well as API endpoints) at the platform's developer console.
2. **Supporting implicit grant:** Alternatively, according to [2, §4.4.2], a unique identifier for the <*authorization endpoint, token endpoint*> tuple can be used as an `issuer` equivalent. However, some integration platforms (*e.g.,* Amazon Alexa [50], Google Assistant [51]) support apps with OAuth implicit grant, which does not involve a token endpoint, rendering this fallback solution inapplicable.[9] Despite that implicit grant is no longer recommended [2, §2.1.2], the necessity for maximizing backward compatibility makes it challenging for integration platforms to adopt this solution.
3. **Customizable access token request:** At least 3 integration platforms we surveyed—Zapier [52], Workato [53] and Bixby [54]—offer greater flexibilities by allowing app developers to write custom code (*e.g.,* JavaScript) to fulfill access token requests (previously discussed in §2.2). Thus the token endpoint is no longer a pre-registered static URL, landing OAuth AS metadata or the <*authorization endpoint, token endpoint*> tuple ill-suited for this use case.
4. **Developer overhead:** The standardized IdP mix-up defense in RFC9207 [3] requires the `issuer` to be returned as a new URL parameter `iss` in the authorization response, which requires extra implementation efforts from *every* AS developer, causing considerable overhead for adoption.
5. **(In)convenience:** From the standpoint of integration platforms, maintaining an app ID is essential for the multi-app system design, regardless of OAuth support. Repurposing the app ID as a defense mechanism for cross-app attacks is thus highly convenient, where extracting and comparing the same identifier from both `state` and `redirect_uri` suffices.

URIs" are susceptible to a mix-up variant ("Cross Social-Network Request Forgery" [49]). Note that this attack is distinct from CORF as it abuses `redirect_uri` matching flaw at AS, which CORF does not assume.

[9] If only the authorization endpoint is used as `issuer` for implicit grant, an attacker could specify a benign app's authorization endpoint and the malicious app's own API endpoint(s) during the registration and development of malicious app. The attack still works.

From the analysis above, we conclude that a per-app ID is the most practical remedy for cross-app OAuth attacks in integration platforms. Additionally, we derive two key insights:

Insight 1: App-centric Architecture. In integration platforms, the layer of *apps* should play a central role in defense:

- In traditional OAuth scenarios, considering the "OAuth client – AS" relationship is sufficient.
- However, in integration platforms, since OAuth registrations are conducted on a per-app basis, this forms a paradigm of "OAuth client – App – AS", representing an app-based isolation boundary. Since multiple apps can share the same `issuer`, a per-app ID is more granular than a per-AS ID (*i.e.,* `issuer`), and should be used for fulfilling both the *functional requirement* (deciding where to send the auth code) and *security requirement* (defending against cross-app OAuth attacks).

Insight 2: Integration Platform Perspective. Based on common functional requirements, an integration platform is designed to support ASes from hundreds or thousands of integrated apps, a scale not considered by IETF specifications.

Given fragmented OAuth support across the numerous ASes, platforms cannot feasibly require *all* ASes to (1) offer OAuth AS Metadata (RFC8414 [24]); and/or (2) return the issuer as a response parameter `iss` (RFC9207 [3]). Otherwise, platforms would have to expel the majority of existing apps.

Therefore, we opt for a countermeasure over which the platform has the most control, driven by platform-generated per-app IDs. In contrast, in traditional OAuth, since only a handful of IdPs (or ASes) need to be dealt with per OAuth client, IETF specifications for mitigating mix-up attacks rely on defenses dependent on the AS (*i.e.,* the `issuer`).

B Practical Impact of CORF

CORF generally has less impact than COAT, as CORF passively logs information while COAT actively acquires privileges. This aligns with the typical severity difference between Account Takeover and Login CSRF vulnerabilities.

However, CORF can still have significant practical implications: 1) In scenarios where the integrated app is used for data synchronization purposes (*e.g.,* syncing calendars, notes, or to-dos), a successful CORF attack could enable wholesale exfiltration of end-user's data. 2) In smart homes, while there may not be an obvious impact from controlling an IoT device not belonging to the victim, CORF can cause a Denial of Service (DoS) of IoT device control by forcefully logging the victim out of their own account at the IoT provider, a side effect of forced account linking. 3) Additionally, CORF can remain largely undetected because most platforms do not notify end-users or display such account replacements in their user interface (UI).