Demystifying the (In)Security of OAuth-based Account Linking in Connector Ecosystems

Kaixuan Luo[†], Xianbo Wang[†], Pui Ho Adonis Fung[‡], Wing Cheong Lau[†]
[†]*The Chinese University of Hong Kong* [‡]*Samsung Research America*

***Abstract*—Modern productivity apps, automation platforms, and AI agents orchestrate across external tools through cloud-based "connectors". To obtain authorized access to connector accounts, these applications rely extensively on the OAuth 2.0 protocol. However, tracking authorization context across web origins and user-agents, while maintaining the binding to the applications' own user identities (*i.e.,* a secure *Account Linking* process), pushes OAuth beyond its original client–server model assumptions. The rise of the OAuth-as-a-Service (OaaS) paradigm further complicates trust boundaries in OAuth.**

In this paper, we present the first comprehensive study of OAuth-based account (mis)linking in connector ecosystems. By systematizing real-world account linking architectural patterns, we show how "OAuth connections", commonly introduced to manage account linking, can inadvertently break session integrity and security boundaries in OAuth. This enables multiple forms of connector account takeovers.

We develop `OASIS` (OAuth Session Integrity Scanner), an analysis framework that identifies account linking implementations and detects novel *Cross-user OAuth session fixation (COSF)* vulnerabilities in mobile apps. Our empirical analysis discovers 40 vendors susceptible to COSF and identifies additional Cross-tenant confused deputy threats in 8 OaaS providers. We propose practical countermeasures that have since been adopted by major vendors such as Amazon Bedrock AgentCore. We lead ongoing discussions and standardization efforts to update OAuth security best practices in the IETF.

1. Introduction

Connector Ecosystem. Modern software systems are transforming how applications (apps) interact with external tools on behalf of users. Productivity apps such as Evernote integrate with SaaS services like Outlook Calendar to sync user data across clouds. Integration platforms [1] such as Microsoft Power Automate, IFTTT, Amazon Alexa, and Google Home enable broader automation with a portfolio of services. OAuth 2.0 [2] has been the predominant protocol for delegated authorization, allowing end-users to grant applications access to the services without password sharing.

The emergence of Agentic AI has accelerated this trend. Developers can now build autonomous agents through infrastructures such as Microsoft Copilot Studio, Amazon Bedrock AgentCore, and Composio. These agentic AI infrastructures introduce the OAuth-as-a-Service (OaaS) paradigm, where a centralized "Token Vault" [3], [4] handles OAuth token retrieval and storage for multiple AI agents.

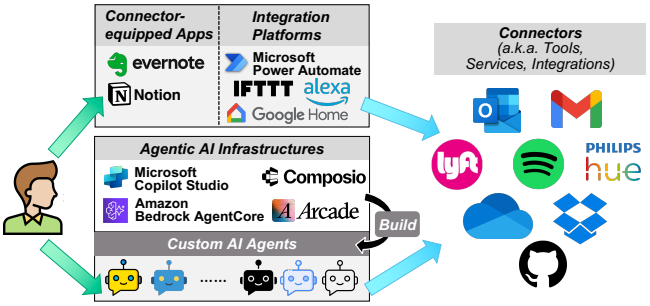

Figure 1. Connector Ecosystems. With OAuth 2.0, users can delegate access to connectors to various types of applications, such as *connector-equipped apps*, *integration platforms*, and *AI agents*. These applications can then invoke the connectors to perform tasks on behalf of users.

In the aforementioned ecosystems, each external service integration, with its OAuth configurations, API endpoints (*a.k.a.* Resource Servers), and wrapper functions for API calls, is encapsulated as a "connector". Collectively, these OAuth-enabled settings define what we term the *connector ecosystems* (Fig. 1). However, applying OAuth securely in this context is non-trivial: OAuth architectures have evolved from simple client–server models into complex patterns executing flows across disparate web origins and user-agents (UAs), and even into multi-tenant architectures as AI agents offload OAuth responsibilities to OaaS providers.

The Missing Piece of OAuth Authorization. Prior OAuth research [5], [6], [7], [8] has primarily focused on traditional applications with simple client–server architectures. As illustrated in Fig. 2, OAuth at the protocol level concerns how an *application (OAuth client)* obtains an end-user's access token from an *authorization server (AS)*, treating the application as a single logical entity. However, OAuth remains agnostic to the "authorization context", regarding how, in a *multi-user application*, the obtained access token from an AS should be associated with the app's own user identity (*i.e.,* "Account Linking").

Once this dimension is taken into account, critical ambiguities emerge. While existing OAuth standards [2], [9] adequately handle same-origin and same-UA scenarios, challenges arise when the application's *user session* is maintained in a different environment from the *OAuth client*, such as on a distinct origin (in web apps), a separate UA (in native apps), or under a different entity (as in OaaS architectures).

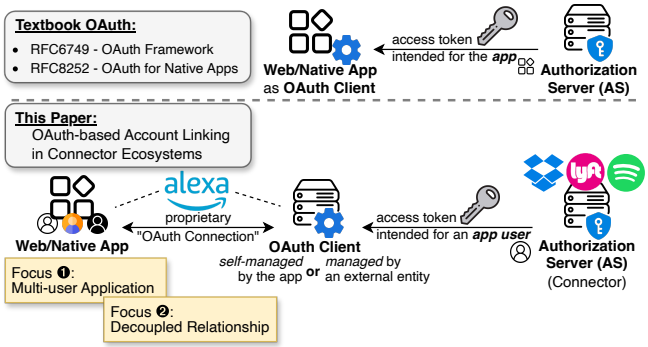

Figure 2. Textbook OAuth *vs.* Account Linking. Textbook OAuth standardizes token negotiation for an application (OAuth client) but is agnostic to its user session management. Account Linking further specifies how an app associates a token with the app's user identity. Notably, the app's core business logic (where user session resides) is often decoupled from its OAuth client component. Account Linking is prevalent in connector ecosystems.

Notably, existing mechanisms like `state` parameter binding [2, §10.12] are underspecified regarding end-to-end session integrity in such cases.

Our Study. In this paper, we conduct a systematic examination of OAuth architectural patterns in connector ecosystems and their security implications. We reveal how implementation details left underspecified by the OAuth protocol play a critical role in accurately and securely modeling account linking flows. At its core, we observe that OAuth clients often introduce an abstract *OAuth connection* to manage a token throughout its lifecycle:

1) It governs token retrieval as an authorization session in OAuth flows (*i.e., an OAuth session*). When user sessions in the UA are inaccessible, applications often resort to OAuth sessions, passed via URLs, to identify end-users.
2) It serves as a persistent handle for internal reference to the token, such as for API calls and token refresh.

Conceptually, an OAuth connection can be characterized by three key elements that form its authorization context: <*user*, *tenant*, *connector*>, representing the app's user identity, the tenant (in OaaS), and the connector involved.

Attacks. The gap between the authorization context tracked by OAuth connections and those enforced in OAuth flows exposes critical vulnerabilities absent under standard OAuth assumptions. Table 1 summarizes our threat analysis.

Cross-user Attacks. Cross-user OAuth Session Fixation (COSF) arises when an OAuth session is used to complete an OAuth flow without verifying its binding to the initiating user's identity. An attacker can pre-establish (fixate) a controlled OAuth session and trick a victim into completing authorization; the resulting token, carrying the victim's connector access, is then linked to the attacker's app account. Unlike traditional web session fixation [10], [11], [12], which hijacks user sessions through unauthenticated session IDs, COSF exploits OAuth sessions derived from existing user sessions to hijack subsequent authorizations.

Cross-tenant Attacks. The threat landscape expands further under OaaS. As a single "Token Vault" serves multiple, potentially untrusted applications (agents) across diverse origins and UAs, the OaaS architecture is inherently susceptible to COSF. It further enables cross-tenant attacks, in which a malicious agent can compromise authorizations of agents in other tenants. These include two confused deputy-style [13] attacks, *Cross-tenant Client ID Confusion* and *Cross-tenant COAT* (Cross-connector OAuth Account Takeover), which exploit the blurred trust boundaries between tenants, and between both tenants and connectors, respectively.

Security Impact. In many cases, a single hyperlink click is the only interaction needed to compromise a user. These attacks enable account takeovers, granting attackers unauthorized access to the connected services. The same exploit can target *any connector* in a vulnerable application, or that within *any tenant* of a vulnerable OaaS provider.

Vulnerability Detection. To facilitate our analysis, we develop `OASIS` (OAuth Session Integrity Scanner), a static analysis framework that identifies account linking implementations and detects COSF vulnerabilities in Android apps using an exclusion-based approach. `OASIS` identified 12 apps supporting account linking and, in an extended dataset, detected 20 vulnerable apps with high accuracy.

Measurement Study. Beyond mobile apps, our broader investigation, combining `OASIS`-backed scans with a curated set of leading vendors, reveals industry-wide threats: 1) COSF vulnerabilities in 19 connector-equipped apps and 15 integration platforms, including top vendors such as Slack, Alexa, n8n [14], IFTTT, Perplexity, and Manus. 2) 19 instances of cross-user or cross-tenant vulnerabilities across 8 agentic AI infrastructures, including Amazon Bedrock AgentCore [3], Microsoft Copilot Studio, and Composio.

Responsible Disclosure. To date, we have received acknowledgments from 20 vendors, with over 15 confirmed fixes and a total of $35,850 in bug bounties. Drawing on this experience, we distill a set of countermeasures, ranging from short-term mitigations to long-term architectural defenses, and lead ongoing standardization efforts to update OAuth security best practices in the IETF, ensuring the safe deployment of OAuth in rapidly evolving connector ecosystems.

Contributions. Our main contributions are as follows[1]:

- We demystify OAuth architectural patterns and security challenges not captured by textbook OAuth (§2) or related work (§8), but fundamental to connector ecosystems (§3).
- We present Cross-user OAuth Session Fixation (COSF), a new class of OAuth attack prevalent in Account Linking, and propose comprehensive defenses (§4). We further extend the analysis to Cross-tenant confused deputy threats in OAuth-as-a-Service environments (§5).
- We develop `OASIS`, a framework for identifying account linking support and detecting COSF vulnerabilities in mobile apps (§6).
- We uncovered vulnerabilities across 42 vendors through our measurement study (§7).
- We actively engaged in responsible disclosure and are working on updates to OAuth security best practices in the IETF [16].

1. We publicly release the artifacts, including the experimental dataset, source code of `OASIS`, and attack demo videos, on our project website [15]: https://mobitec.ie.cuhk.edu.hk/connector-oauth-security.

Table 1. COMPREHENSIVE OAUTH THREAT ENUMERATION IN "CONNECTOR ECOSYSTEMS". THESE ECOSYSTEMS INTRODUCE AN ABSTRACT "OAUTH CONNECTION" TO MANAGE ACCOUNT LINKING, WHICH TRACKS THE AUTHORIZATION CONTEXT AS A TUPLE OF <*user*, *tenant*, *connector*>.

Category [§3.1]	**OAuth Client**	**OAuth Connection's Authorization Context:** <*user*, *tenant*, *connector*>	**Potential Account Mislinking Threats** Cross-*user* **OAuth Session Fixation** [§4.1] Defense: User Session Binding[a]	Cross-*tenant* **Client ID Confusion** [§5.1] Defense: Tenant Scoping[b]	Cross-*connector* **OAuth Account Takeover** (COAT) Defense: Connector Distinction[c]
Connector-equipped Apps	Self-managed	`<many,1,handful>`	● (Cross-Origin/UA)	○	○ (No untrusted connector)
Integration Platforms	Self-managed	`<many,1,many>`	● (Cross-Origin/UA)	○	● (Single-tenant) COAT [1]
Agentic AI Infrastructures	Managed	`<many,many,many>`	● (Separate Responsibilities)	●	● Cross-tenant COAT [§5.2]

Notation: ● indicates the threat is applicable; ○ indicates the threat is not applicable.
[a] User Session Binding: *User that Starts OAuth == User that Completes OAuth*, enforced via session integrity checks at the original origin or UA.
[b] Tenant Scoping: *Tenant that User Consents == Tenant that Receives Access*, enforced via per-tenant OAuth client registration.
[c] Connector Distinction: *Connector that Starts OAuth == Connector that Completes OAuth*, enforced via globally unique per-connector `redirect_uri`.

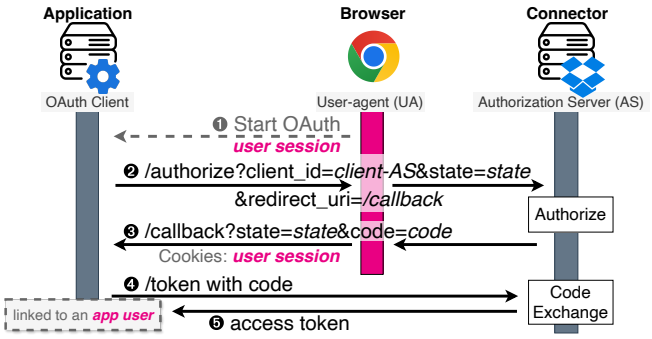

Figure 3. OAuth 2.0 Authorization Code Grant Flow

2. Background: Textbook OAuth

OAuth Authorization Code Grant. The textbook OAuth flow (Fig. 3) follows a "three-legged" design, as it involves three entities: an *OAuth client* hosted by the application in need of tokens; a *user-agent (UA)* for end-user interactions, typically a web browser; and an *authorization server (AS)*, from which the application obtains access tokens. As a running example, a web app (*e.g.,* an online PDF editor) may request access to a user's private files in a cloud storage service (*e.g.,* Dropbox), where the service runs an AS.

An OAuth flow consists of two phases: *authorization* and *code exchange*. The authorization phase would issue a short-lived authorization code (auth `code`): ❶ The user initiates OAuth via the UA, which sends a request to the web app's backend (hereafter, the *OAuth initiation request*). ❷ The user is then redirected to the AS in an authorization request, prompted to log in at the storage service's website and grant authorization consent. Note that explicit consent may be skipped if prior consent exists, making the authorization an automatic process. ❸ An auth code is issued by the AS in an authorization response, passed through the UA, and handed to the web app's backend. It reaches the callback (*a.k.a.* redirection) endpoint, as specified by the `redirect_uri` parameter (redirection URI) in the earlier request.

In the code exchange phase, the OAuth client at web app's backend exchanges the auth code for an access token via a server-to-server request to the AS (token request ❹ and response ❺). Eventually, the access token is associated with the web app, which can then call the service's APIs to retrieve protected resources such as the user's private files.

OAuth Client Registration and Client Types. At steps ❶ and ❹, the AS can uniquely identify the client using a `client_id`. At step ❹, the AS can further authenticate the client using a credential, typically a shared secret (*i.e.,* `client_secret`). The process of negotiating such identifiers and credentials prior to OAuth flows is called *client registration*. A web app running on a backend server can securely store secrets and is therefore referred to as a *confidential client*. In contrast, native apps that store credentials on-device cannot maintain their confidentiality and are treated as *public clients*.

OAuth Session Integrity. In this work, we define OAuth session integrity based on [5], as the property that ***a user who initiates an OAuth flow must be the same user who completes OAuth.*** To date, Cross-site Request Forgery (CSRF) is the most widely-discussed session integrity threat in OAuth. According to RFC requirements [2, §10.12] and [17, §5.3.5], the OAuth `state` parameter, an opaque value sent in the authorization request and returned unchanged in the authorization response should be bound to the UA session. Without proper validation, an attacker can inject an auth code into a victim's unsolicited OAuth flow in step ❸, thereby breaking session integrity and allowing the victim to access attacker's resources (*i.e.,* Login CSRF).

3. Understanding Account Linking

This section demystifies account linking in connector ecosystems, examining the architectural patterns that enable it and the implementation challenges that make it non-trivial. For ease of understanding, Table 2 provides a mapping between standard OAuth roles and common industry jargon. For example, we use the umbrella term *connector* to denote the entity that plays the AS role in OAuth.

3.1. Connector Ecosystems

In ecosystems such as *application integration* [18], [19] and *agentic AI* [20], each external service integration is encapsulated as a *connector*. Applications perform OAuth-based account linking and maintain a persistent "OAuth connection" to that service. This allows applications to perform tasks on behalf of users with authorized access. We scope our study to three representative product categories:

Integration Platforms. Cloud-based integration platforms orchestrate interactions across multiple connectors from an open marketplace, often in a low-code or no-code fashion. Following prior work [1], we examine 3 major types of

Table 2. TERMINOLOGY MAPPING TABLE

OAuth Roles [RFC6749]	Industry Jargon
Authorization Server (AS) & Resource Server (RS)	**Connector** (*e.g.,* Dropbox)
Application (App) *subsumed under OAuth Client*	- **Connector-equipped App** (*e.g.,* Evernote); - **Integration Platform** (*e.g.,* Amazon Alexa); - **Agents** deployed with Agentic AI Infrastructure (*e.g.,* with Amazon Bedrock AgentCore): the OAuth-as-a-Service paradigm. Concrete form as a web or native application.
OAuth Client	- A **Backend Component** of Connector-equipped App/Integration Platform; - **"Token Vault"** in Agentic AI Infrastructure.
Resource Owner	**Connector Account** of the end-user (*e.g.,* Dropbox account)
Not Modeled	**App Account** of the end-user (*e.g.,* Amazon Alexa account)
Not Modeled	**Tenant.** Definition: A tenant is an organization signing up at OaaS to develop custom agent(s) paired with connectors, deploying the agent(s) in its own trust domains for its own end-users.

platforms, including *workflow automation platforms*, *virtual assistants*, and *smart homes*.

Connector-equipped Apps. Applications such as Evernote do not fit neatly into common integration platform types, yet they have specific integration needs. For instance, Evernote supports calendar synchronization through Google and Outlook Calendar integrations. We refer to such apps with a few purpose-specific connectors as *connector-equipped apps*.

Agentic AI Infrastructures. Emerging platforms such as Amazon Bedrock AgentCore provide infrastructures for building AI agents that integrate with connectors. Historically, niche infrastructure-level solutions have existed in the form of *Integration Platform as a Service* (iPaaS) and modern *API Management Service / API gateway*. By contrast, such infrastructures have become pivotal in agentic AI:

AI agents often need to accommodate customized enterprise and organizational requirements, and therefore vendors position themselves as *Agent Builders* or *AI Gateways* to support the development of custom AI agents. End-users interact with agents across disparate trust domains rather than through a centralized integration platform. A typical offering includes a software development kit (SDK) and developer console for building agents and equipping them with pre-built or custom connectors. Agent developers rely on the OAuth client (so-called "Token Vault") provided by the infrastructure for token retrieval and lifecycle management.

From an OAuth perspective, this has led to a new business model, *OAuth-as-a-Service* (OaaS), where OAuth client responsibilities are effectively outsourced to a centralized entity. This also shifts OAuth from the traditional single-tenant model, where each OAuth client serves a single application, to a *multi-tenant architecture* in which a tenant corresponds to one or more agents built by the same organization's developers.

3.2. Account Linking: Second-order OAuth

Following the definition in [1] (with aligned terminology), we define *Account Linking* as the process of ***connecting a user's connector account to their app account.*** Access to the connector account is represented by the OAuth access token (and/or refresh token [2, §6]), while the app account reflects the user's identity within a multi-user application, typically represented by an authenticated user session at runtime. In certain cases (*e.g.,* agents as customer service chatbots), the concept of an app account may extend to cover an anonymous session, which we subsume under "user session". Depending on the use case, OAuth tokens may be either linked to the app account for long-term access, or valid for the lifetime of the current user session.

Textbook OAuth: Track by Session. Revisiting the textbook OAuth flow, the OAuth client may rely on the application's user session stored in the browser for account linking. Yet, the dotted steps in Fig. 3 are outside the scope of the OAuth protocol, namely, how OAuth is initiated and how the application identifies which user should receive the tokens.

Reflecting on Fig. 2, this gap exists because RFC6749 [2] only specifies "first-order OAuth", regarding how an *application* obtains OAuth tokens from an end-user (*i.e.,* the resource owner). Account linking, however, represents "second-order OAuth": an *application's own user* needs tokens from the same user. Textbook OAuth overlooks this distinction because it typically assumes a same-origin, same-UA deployment, where the application and its OAuth client are deployed in a single origin and UA, and therefore the UA (web browser) naturally preserves the user session at the callback endpoint. In more complex architectures where the application and its OAuth client are decoupled components, this assumption no longer holds. To better reflect this decoupling, we hereafter use the term *OAuth client* to refer specifically to the component that handles OAuth responsibilities of the application, and the term *application* to refer to its general business logic (*e.g.,* user session management).

3.3. Decoupled OAuth Architectural Patterns

Modern OAuth deployments often violate textbook OAuth assumptions to support account linking across web origins, UAs, and security boundaries. Such decoupled OAuth architectural patterns create three key challenges:

Challenge 1: Cross-Origin Session Tracking. When account linking spans multiple origins in a web application (see Fig. 4), for instance, account linking initiated at origin A but with the OAuth callback handled by origin B, origin B cannot determine which user the received authorization should be associated with. This occurs because the user session at origin A is not accessible to origin B due to same-origin policy [21] (or more precisely, due to site isolation at the registrable domain level, *i.e.,* eTLD+1 [22]). This situation is common in modern deployments where server-side components are decoupled, such as a shift from monolithic to microservices architecture.

Challenge 2: Cross-UA Session Tracking. As depicted in Fig. 5, when a native app (*e.g.,* an Android mobile app) supports account linking and follows RFC8252 [9] to use an external UA (web browser) to handle OAuth, it launches the browser to request authorization for the connector. However, due to process isolation, the native app cannot share the user session with the browser, thereby breaking session tracking between the two UAs and creating a session gap at the OAuth callback. To date, OAuth specifications lack guidance

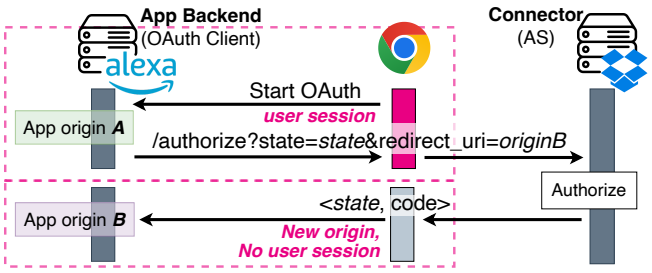

Figure 4. Challenge in Cross-Origin OAuth: Server-side decoupling separates application components. However, the browser cannot track user session across origins due to Same-origin policy (SOP) and site isolation.

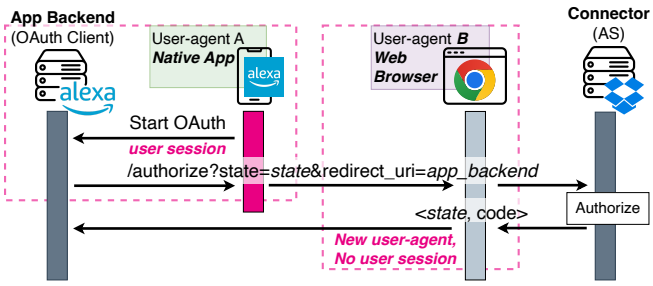

Figure 5. Challenge in Cross-UA OAuth: Native apps follow RFC8252 to perform OAuth in an external UA (browser). However, the browser cannot track the native app's user session due to process isolation.

for native apps acting as confidential clients. RFC8252 [9] only covers native apps as public clients. Yet, during account linking, a native app behaves more like a UA, with its OAuth client implemented in the backend as a confidential client.

Challenge 3: Outsourced OAuth Responsibilities in OaaS. Custom AI agents (applications) outsource OAuth responsibilities to the OaaS, where its "Token Vault" acts as an OAuth confidential client to negotiate OAuth tokens with the connector's AS, and securely stores them for the agents. Agents typically authenticate to the Token Vault using per-tenant API keys [23] or per-agent workload identity tokens [24], [25] to retrieve the tokens or make API calls, thereby avoiding the complexity of implementing OAuth client logic and token management themselves.

As illustrated in Fig. 6, agents that rely on OaaS inherently cross origins and/or UAs, since their main business logic (including user sessions) remains in the tenants' own web origins or native apps, whereas the OAuth callback is operated by the Token Vault. This situation is further complicated by the heterogeneous agent publishing *channels*: To interact directly with end-users, agents may be deployed across various environments, ranging from web/native apps to bots within instant messaging (IM) software such as WhatsApp and Telegram (see Fig. 7). While web and native apps are fully controlled by the agent developer, IM channels depend on external application contexts, compounding the session tracking challenge. Consequently, a session gap exists not only between the *Token Vault* and the *agent*, but may also independently exist between the agent's *backend* and its *channel* (the UA running its frontend).

Expectations. Despite these challenges, decoupled OAuth deployments should uphold the following expectations:

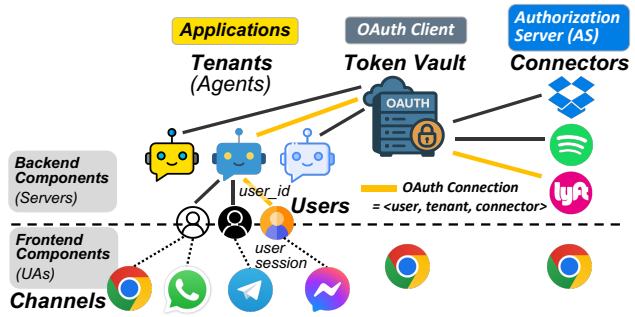

Figure 6. OAuth-as-a-Service in Agentic AI Infrastructures: Agents can be deployed in any channels, crossing origins and UAs by design.

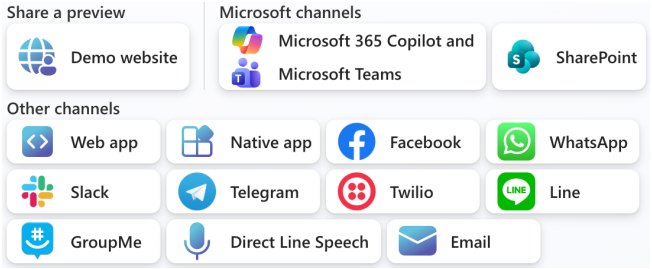

Figure 7. Channels for Publishing an Agent in Microsoft Copilot Studio

- From the *application's perspective*, upon receiving a user's request to initiate OAuth, it queries the OAuth client to generate an authorization URL, returns it to the user, and waits for the OAuth client to obtain the token. After OAuth completes, the app may access protected APIs in one of two methods: (1) typically by calling the OAuth client, which acts as an API gateway that routes requests to the appropriate connector's APIs with the correct token attached, or (2) less commonly by retrieving the raw access token from the OAuth client, allowing the app to invoke the APIs directly, as seen in some OaaS providers.
- From the *OAuth client's perspective*, it needs to fulfill the application's request to complete OAuth flows, while keeping track of the *context* in which the application requests authorization.
- From the *end-user's perspective*, the user experience should be consistent with textbook OAuth. The proprietary details between the application and decoupled OAuth client should be opaque to users.

3.4. OAuth Connection

To coordinate between the application and the OAuth client, vendors commonly introduce the abstract notion of an ***OAuth connection***, which uniquely identifies each OAuth token together with its authorization context in account linking. While a token represents access to a protected resource, the OAuth connection manages an application's access to that token throughout its lifecycle. To identify the authorization context, each OAuth connection is associated with certain metadata. Formally, we define the authorization context of an OAuth connection as a tuple:

OAuth connection = <user, tenant, connector>.

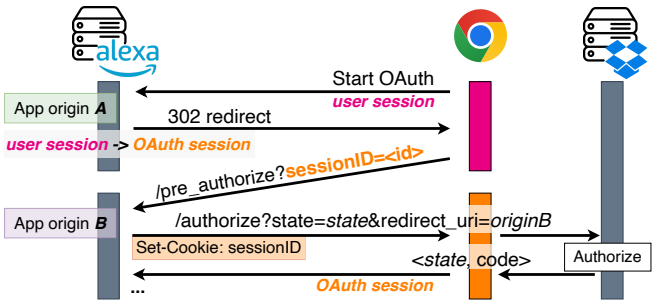

Figure 8. OAuth Session as a standalone parameter in URL

At a minimum, the OAuth client should track the recipient user of the token, and may also track the current connector or tenant if it supports multiple connectors or tenants. An end-user may also configure multiple OAuth connections for the same connector at a tenant (*e.g.,* to manage multiple Gmail accounts), in which case additional distinguishing metadata may be included, keyed by a unique connection ID.

OAuth connection is used both during OAuth flows and for token management afterward. (1) During an OAuth flow, the OAuth connection is instantiated as an authorization session (hereafter, *OAuth session*), which allows the application to track the connection in the user's browser. OAuth session is derived from and maintained independently of the application's user session. (2) After OAuth completion, the OAuth connection serves as the handle to reference the obtained token, such as in token refresh and API calls.

3.5. Common (but Flawed) Designs

In practice, we observe two common implementation patterns for tracking OAuth connections in OAuth flows: embedding an OAuth session identifier (*OAuth sessionID*) in the URL, either as a standalone, client-defined parameter, or within the OAuth `state` parameter.

OAuth Session in standalone URL parameter. A common pattern inserts an additional request between the OAuth initiation request and the authorization request. This additional *pre-authorization request* carries the OAuth sessionID as a URL parameter to transfer session information from origin A to origin B, or from a native app to a web browser. As an example, in a cross-origin scenario, when the server at origin B receives the pre-authorization request, it redirects the user to the authorization endpoint and either sets the OAuth session in the browser (*e.g.,* via Set-Cookie [26], as illustrated in Fig. 8), or continues carrying the session value in the URL (*i.e.,* via the `state` parameter).

OAuth Session in `state` parameter. An alternative design embeds the OAuth session identifier directly into the `state` parameter. As an example, Fig. 9 shows such a design under cross-UA setting. This approach is appealing because, per OAuth specification [2, §4.1.1], the `state` parameter is intended to maintain state during the authorization phase.

Unfortunately, the above common designs are also inherently insecure. They allow the OAuth session to be fixated within any UA, exposed to any origin, and propagated into the OAuth callback, which introduces serious security risks.

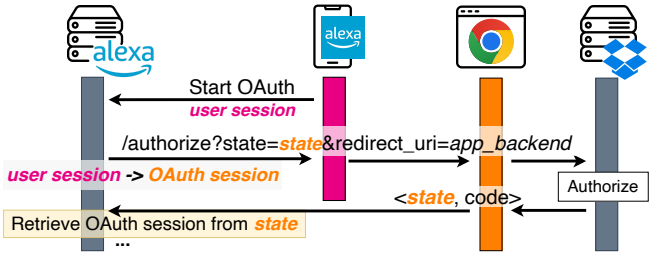

Figure 9. OAuth Session in the `state` parameter

3.6. Threat Model

Our study considers two threat models that build upon the *web attacker* model [27], [28]. They capture two realistic ways in which an untrusted component can infiltrate connector ecosystems: either as an application's *end-user*, or as a *developer* (via an OaaS tenant). This enables five new types of attacks spanning two classes (cross-user and cross-tenant attacks) within connector ecosystems. In both models, the victim is an honest user of a benign application. These attacks do not require an active user session of the *application* in the victim's browser; however, an existing user session on the *connector* side can enhance attack stealthiness.

Basic Model: Untrusted User (Three Cross-user Attacks in §4).

- *Attacker Capabilities.* The attacker is a regular user of a benign application with no special privileges.
- *Attack Scenario and Goal.* The attack occurs in a cross-user scenario and typically involves the victim clicking a crafted link provided by the attacker (§4.1, §4.2.1). By doing so, the attacker can hijack access to the victim's connector by linking it to attacker's app account. Some variants (§4.2.2) require no user interaction: the attacker can forcibly bind their connector account to the victim's app account, causing victim to use attacker's resources.

Extended Model: Untrusted Tenant (Two Cross-tenant Attacks in §5). Under OaaS, tenants that are otherwise well-isolated may lack isolation at the OAuth protocol level. Tenants reflect a realistic organizational security boundary:

- *Attacker Capabilities.* An attacker can freely sign up for the OaaS to set up a malicious tenant.
- *Attack Scenario and Goal.* A malicious tenant can trick the victim into interacting with a connector associated with the attacker's application. This poses cross-tenant threats, where the victim's connector access in a benign tenant can be compromised by the malicious tenant.

4. Cross-user Attacks

The victim *completes* an OAuth flow, but may not be the one who *initiated* it. Crucially, under the common design patterns in §3.5, the initiator of the OAuth flow is the user to whom the authorization is linked. Consequently, if a malicious user initiates an OAuth flow that a victim later completes, the victim's connector account would be linked to the attacker's app account. We refer to such account mislinking issues in cross-user settings as *Cross-user Attacks*.

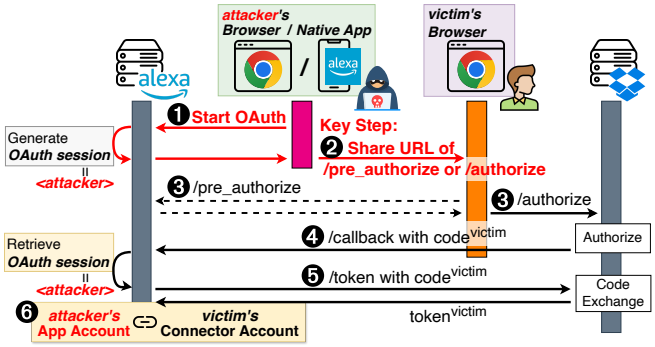

Figure 10. Attack Flow of Cross-user OAuth Session Fixation (COSF)

4.1. Cross-user OAuth Session Fixation (COSF)

4.1.1. Attack Flow. In a COSF attack, the attacker fixates an OAuth session in the victim's browser to hijack the connector access. Assume the victim has previously linked the targeted connector account to their app account and that a valid session for the connector account exists in victim's browser. The attack steps are illustrated in Fig. 10, and outlined as follows:

❶ The attacker initiates account linking with the targeted connector via the application.

Variant 1: The attacker records the *pre-authorization request URL* (*e.g.,* `https://client.com/pre_authorize?sessionID=7dpf1jhpq4q3s5fnj0hts1oou8`), where the `sessionID` value is the OAuth session identifier derived from the attacker's app identity.

Variant 2: The attacker records the *authorization request URL* (*e.g.,* `https://connector.com/authorize?client_id=<targeted_connector>&redirect_uri=https://client.com/callback&state=7dpf1jhpq4q3s5fnj0hts1oou8`), where the `state` encodes the OAuth session.

❷ The attacker shares the pre-authorization request URL (Variant 1) or the authorization request URL (Variant 2) with the victim as a hyperlink.

❸ When the victim clicks the link, their browser would be redirected to the app's authorization endpoint (Variant 1) or would directly visit it (Variant 2).

❹ If prior consent exists, the access is granted automatically, with `code` and `state` sent back to the app's OAuth callback. While some cookies may be attached, the app backend only consumes the cookies used to maintain the OAuth session and/or to validate `state`; the app's session cookies are not available or not used to verify user identity.

❺ The app's OAuth client exchanges victim's auth code for an access token and associates this token with attacker's app account, as indicated by the fixated OAuth session.

❻ As a result, the *victim*'s connector account is linked to the *attacker*'s app account. The attacker gains unauthorized access and can control the connector on the victim's behalf.

4.1.2. Security Impact.

Account Takeover. The direct impact is *unauthorized access*. Notably, as connector authorizations commonly grant broad scopes for access delegation that resemble user impersonation, these attacks essentially amount to *account takeovers*. For example, the attacker compromises the victim's Dropbox connector, then the attacker is able to access the victim's Dropbox files directly from the attacker's app account. Moreover, since the weakness lies in the *OAuth client*, which serves all connectors as a single, shared entity:

- In *connector-equipped apps*, although only a handful of connectors are supported, all of them are at risk.
- In *integration platforms*, all connectors in the platform's marketplace are equally affected.
- In *agentic AI infrastructures*, vulnerabilities extend to all tenants and every connector used by their agents.

Consequently, as long as the victim is an end-user at any of the affected vendors, they risk having their Dropbox connector access taken over, facilitating targeted attacks.

One-click Attacks. Connectors such as GitHub, Dropbox, Outlook Calendar, and OneDrive support silent/automatic authorization. This means that when prior consent is in place (*i.e.,* if the victim user has previously connected these connectors with the application under a pre-configured `client_id`), no explicit consent would be required for another linkage, enabling stealthy, one-click attacks.

Practical Consent Phishing. For connectors that always require explicit consent or for first-time authorizations, the threat remains and still constitutes a valid attack. This is because the victim has no way of knowing that their connector account would be linked to the attacker's app account rather than their own. The OAuth consent screen presents the token recipient as a *trusted entity* (which is the app itself, *e.g.,* Amazon Alexa), making these attacks more convincing than conventional consent phishing campaigns in OAuth [29].

Attack Comparison. In classical web session fixation [10], [11], [12], an attacker fixates a session identifier in the victim's browser to hijack their user session. In Cross-user OAuth Session Fixation, the attacker fixates an OAuth session identifier derived from the established user session to hijack the victim's connector access.

4.2. COSF-related Implementation Flaws

This section discusses two implementation issues in account linking that rely on COSF, where OAuth sessions either jeopardize the app's user sessions or are guessable.

4.2.1. Login CSRF of App Account. When the OAuth session is initially embedded in a standalone URL parameter (§3.5), we noted that a dedicated session cookie is typically set to track the OAuth session at the callback. In some apps, however, this cookie instead tracks the full *user session* of the account linking initiator. As a result, if the pre-authorization request URL is shared with a victim, an attacker can leverage the fixated session to break the integrity of the application's user session, resulting in Login CSRF of the app account. Meanwhile, the OAuth flow proceeds uninterrupted, facilitating connector account takeovers via COSF, effectively "killing two birds with one stone".

4.2.2. 0-Click Forced Linking of Connector Accounts. The attacks described so far require user interaction. How-

ever, if the OAuth session (ID) is guessable and its integrity is not protected (*e.g.,* merely consisting of an 8-digit user ID, or containing such an ID in a non-signed JSON), forced linking of connector accounts can occur without any victim interaction. An attacker can forge an OAuth session (ID) and then submit their own auth code at the OAuth callback. This forces the victim associated with the targeted user ID to access the attacker's connector account at the application.

4.3. Robust Defenses

Root Cause. To summarize, cross-user attacks arise because the OAuth client relies on a derived *OAuth session*, instead of the app's existing *user session*, to identify the intended "app account" for account linking. However, it fails to verify the binding between the two sessions, likely due to the OAuth callback being visited in an *unauthenticated space*, a distinct origin or UA separated from the original *authenticated space* holding the app's user session. Malicious users can exploit this lack of binding to transfer OAuth sessions to a victim's device, enabling session fixation attacks.

To maintain session integrity during account linking, the guiding principle is to ensure that **an OAuth flow *initiated* by one app user is also *completed* by the same user**. Practically, this means having the OAuth flow complete at an authenticated space, namely, the app's primary web origin or its native app. Robust defenses therefore either directly return to the original origin or UA (D1 and D2), or apply a post-redirect pattern that securely bridges the gap between origins and UAs (D3). These approaches eliminate insecure session passing, ensuring that any passed session is routed back and validated in the authenticated space.[2]

D1. Web App: Cross-Origin Defense. The application shall register its OAuth callback endpoint under the same origin where its user session resides. Upon OAuth callback, the app's backend *must* verify the OAuth session against the user session to ensure that the entire OAuth flow is both initiated and completed by the same app user, before performing the code exchange. Note that if the "real" OAuth callback processing logic remains on a cross-origin location, the app should forward the auth code to the location from its *backend*, rather than exposing a cross-origin callback to the *frontend* (UA) as in the vulnerable design.

D2. Native App: Cross-UA Defense. Native apps should register their callback endpoint as a native public client [9, §7] at the connector's AS. Upon receiving the authorization response, the native app should submit the auth code to the OAuth confidential client in the app's backend. Implementations *must* verify the OAuth session against the user session before proceeding with code exchange.

D3. Post-redirect Pattern. There are cases where changing the preregistered callback endpoint on the connector side is impractical: for example, when securing an existing implementation that requires a quick fix, or for a cross-platform app where a single `client_id` serves both its web and native versions. In such cases, a *post-redirect pattern* can be employed: the OAuth callback acts as a proxy that forwards the auth code and `state` back to the app's original origin or UA where account linking is initiated.

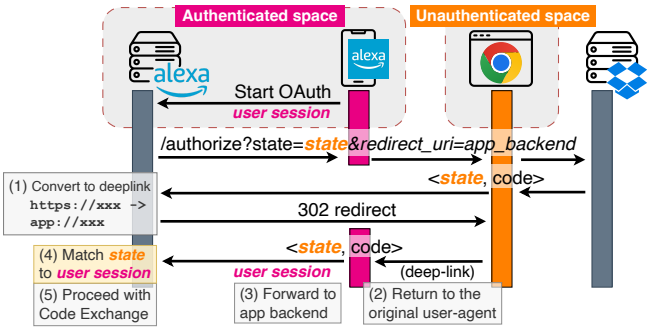

Figure 11. Robust Defense: Return to and validate at the authenticated space (app's original origin/UA with user session) maintains session integrity.

As an example, the defense in native apps is illustrated in Fig. 11. The callback endpoint converts the request URL it receives into a deep link and redirects to it, returning control to the native app. The app then sends the auth code to its backend and verifies that the logged-in user indeed initiated the OAuth session before continuing the code exchange.

Defense Comparison. Traditional session fixation in web security has a well-established defense by rotating session IDs [10], which is now standard practice in web applications. In contrast, defending against COSF requires verifying the binding between the OAuth session and the existing user session, which is often overlooked.

4.4. Other Valid Countermeasures

In addition to the recommended defenses (**D1–D3**), we further outline several additional viable defenses (**D4–D6**) with their respective limitations.

D4. Manually Bridging the Session Gap. Given the heterogeneous publishing channels for agents, their ability to handle OAuth callbacks varies. While web and native apps can host callbacks directly, agents deployed in constrained environments, such as bots running on IM platforms, face greater challenges. Although callbacks to the IM platform itself are feasible (*e.g.,* via HTTPS URLs or deep links), bots typically have limited capability to receive such callbacks. This limitation calls for alternative approaches to the post-redirect pattern. One option is for the OAuth client to display a PIN code that the user manually enters, similar to mechanisms used to mitigate phishing attacks in cross-device OAuth flows [30, §3.1.3]. The PIN can then be relayed through the channel to app (agent) backend and ultimately returned to the OAuth client for validation. Alternatively, connector's AS may directly employ cross-device flows instead of authorization code grant, as we move toward even more heterogeneous agent interfaces like headless agents.

D5. Extra Consent Screen at OAuth Client. An alternative defense is to require explicit user consent at the

2. Note that the defenses D1–D3 still assume the existence of an *OAuth session*. Under the same principle of returning to the authenticated space, if vendors instead abandon OAuth sessions and rely solely on the app's *user session* for user identification at the OAuth client, the session integrity problem reduces to standard OAuth Login CSRF in §2.

pre-authorization request or OAuth callback. This additional consent screen, enforced by the OAuth client, shall display which *app user* would receive the OAuth token. It complements the original OAuth consent screen at the AS, which shows which *application* would receive the OAuth token.

D6. Re-authentication in the Browser. For native apps, a straightforward defense is to require users to re-authenticate the application in the browser, either at the pre-authorization request or OAuth callback. This would turn the cross-UA scenario into a same-UA scenario. However, this approach degrades user experience and may cause confusion.

Limitations.

- Defenses that depend on end-user awareness and manual decision-making remain vulnerable to *phishing attacks*. These include the manual session-transfer method (D4) and the extra consent mechanism (D5).
- The defenses D5 and D6 are limited in that the OAuth client cannot present meaningful user information on the consent screen, or enforce re-authentication, when the application's user session is not tied to a logged-in user account but rather an anonymous session.

Other Defenses. Beyond the basic defenses, Appendix A.1 describes tailored post-redirect pattern (D3) protections for OaaS. Appendix A.2 summarizes common pitfalls and ineffective defenses, such as PKCE's [31] inability to mitigate the issue and additional open redirect [32] risks introduced by improper fixes.

5. Cross-tenant Attacks in OAuth-as-a-Service

So far, our analysis focused on scenarios where the application and its OAuth client are maintained by the same party. In OaaS, however, the centralized OAuth client (Token Vault) maintains a *third-party* relationship with potentially untrusted applications (agents) from different tenants. This multi-tenant architecture expands the attack surface and enables new cross-tenant confused deputy attacks.

In such a setting, while the OAuth connection suffices for the *functional requirement* of tracking the tenant and connector, the *security requirement* further relies on mechanisms native to the OAuth protocol, namely the client ID and redirection URI, to distinguish them. These identifiers cannot be scoped only at the coarser per-OAuth-client level.

5.1. Cross-tenant Client ID Confusion

Problem Definition. OaaS providers aim to reduce the OAuth integration burden for agent developers. Beyond token lifecycle management, many providers offer pre-built connectors for popular services (*e.g.,* Dropbox, Outlook Calendar, GitHub) with preconfigured APIs. To support these tools out of the box, some OaaS providers also handle OAuth client registration for these connectors, with `client_id` and `client_secret` baked in. Traditionally, each developer must register their application at the AS to obtain a unique client ID, which identifies the application during OAuth flows. In OaaS, however, multiple tenants may share the same built-in client ID assigned to the Token Vault.

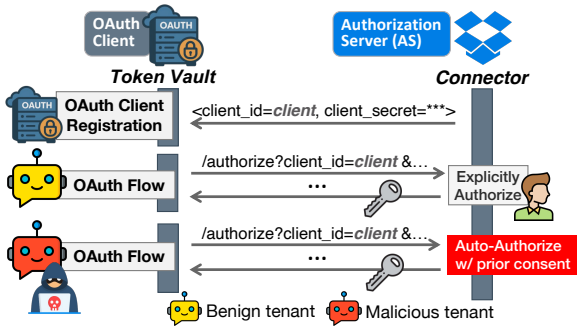

Figure 12. Cross-tenant Client ID Confusion. A user previously consented at a benign tenant under a shared `client_id` of a pre-built connector. Connector access granted without consent to an attacker-controlled tenant.

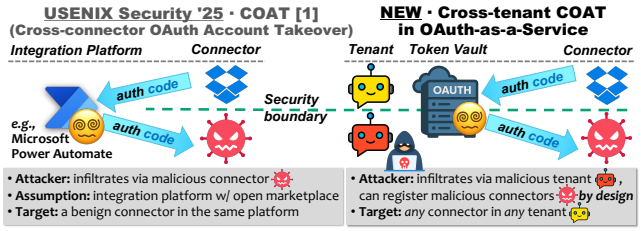

Figure 13. Comparison of (Single-tenant) COAT and Cross-tenant COAT

Attack. This design introduces the risk of *Client ID Confusion*. As shown in Fig. 12, consider a user who previously authorized a benign connector for a trusted agent belonging to a benign tenant. Although the resulting access token is scoped to that tenant, the user's consent is technically granted to the Token Vault's shared client ID. Consequently, when the same user later interacts with a malicious agent that also relies on the Token Vault, the benign connector treats the new authorization request as pertaining to the already-consented client ID. As a result, access is silently granted to the malicious agent, contrary to the user's intention to withhold or explicitly consent to the authorization.

Countermeasure: Per-tenant OAuth Client Registration. The root cause of Client ID Confusion is that as multiple tenants share the same client ID at a connector, a malicious tenant can inherit user consent previously granted to a benign tenant. To prevent this issue, OaaS providers *must not* supply shared client IDs for tenants in production environments. Each tenant should instead register its own dedicated client ID with the connector's AS and bring it to the OaaS ("Bring Your Own `client_id`", BYOC). This ensures that authorization consent is always tied to the intended tenant and avoids cross-tenant unauthorized access.

5.2. Cross-tenant COAT Attack

Problem Setting. Prior work [1] introduces the Cross-connector OAuth Account Takeover (COAT) attack[3], where an integration platform, acting as a confused deputy, can

3. Originally referred to as "Cross-*app* OAuth Account Takeover" in [1], where the term "app" denotes SaaS applications integrated within a platform, and thus corresponds to the "connector" concept in this paper.

leak a victim's auth code from a benign connector to a malicious one. In the original setting, the attacker relies on a single tenant—an integration platform serving an *open marketplace*—to distribute the malicious connector. With multi-tenant OaaS, this prerequisite is relaxed.

Attack. As illustrated in Fig. 13, in OaaS, any developer can create custom agents paired with custom connectors within their own tenant. Consequently, the presence of a malicious connector no longer depends on an open marketplace, but remains applicable even if the targeted benign tenant(s) are *closed ecosystems*: an attacker can simply deploy a malicious agent that carries a malicious connector in their own tenant, and target users of a benign connector carried by a benign tenant, launching *Cross-tenant COAT* attacks.

In an attack flow, the victim establishes account linking with the malicious connector (as a dummy user of the malicious agent), and is then redirected to authorize the benign connector (as a user of the benign agent). However, the resulting auth code is forwarded to the malicious connector in the token request because the OAuth client, based on the OAuth session, tracks the malicious connector in the authorization context. The attacker can then redeem the stolen auth code at the benign connector (as a user of the benign agent) for connector account takeover.

Countermeasure: Globally-unique `redirect_uri`. In the original (single-tenant) COAT vulnerability [1], the OAuth client fails to verify that the connector completing OAuth matches the one that initiated it. In Cross-tenant COAT, the OAuth client (Token Vault) further fails to ensure that both connectors belong to the same tenant. To address this root cause, the OAuth client shall assign a *globally unique* `redirect_uri` for each connector during client registration and OAuth flows. This design enables the OAuth client to enforce `redirect_uri` validation not only between connectors within the same tenant, but also across tenants. The exact implementation of the `redirect_uri` may vary; for example, using a globally unique connector ID (*e.g.,* `https://oaas.com/callback/<globally-unique-ID>`), or combining a tenant ID[4] with a tenant-scoped connector ID (*e.g.,* `https://<tenant>.oaas.com/callback/<intra-tenant-unique-ID>`).

Comparison: Cross-user vs. Cross-tenant Attacks. Recall from §3.4 that an OAuth connection tracks the authorization context of <*user*, *tenant*, *connector*>. The aforementioned attacks exploit different gaps in this context: Cross-user attacks break the *user* binding; Cross-tenant Client ID Confusion breaks the *tenant* scoping; and Cross-tenant COAT breaks the *connector* distinction.

6. COSF Vulnerability Detection

In this section, we propose `OASIS` (OAuth Session Integrity Scanner), a static analyzer that evaluates the prevalence of COSF vulnerabilities across connector ecosystems. This section presents its design and implementation.

4. Note that enforcing per-tenant `redirect_uri` alone mitigates Cross-tenant COAT, reducing the attack surface to Single-tenant COAT [1].

6.1. System Overview of `OASIS`

Evaluating COSF vulnerabilities across the industry is non-trivial in two aspects: **1) Identifying in-scope applications.** While common *integration platforms* and *agentic AI infrastructures* can be readily identified due to their inherent reliance on account linking, *connector-equipped apps* are more difficult to discover, as account linking appears as a secondary feature within broader productivity apps. **2) Assessing vendor susceptibility** further requires extensive manual testing, since no reliable automated detection methods exist. These gaps motivate the development of a tool to uncover COSF vulnerabilities.

A major challenge is that COSF vulnerabilities manifest in OAuth confidential clients deployed in the cloud, whose source code is not publicly available, which precludes direct static analysis. Our key insight is to shift focus to local codebases instead. Rather than analyzing *vulnerability patterns* in cloud-side implementations, we adopt an **exclusion-based approach** that identifies *the absence of certain patterns* in locally-accessible Android app code, which indirectly signals potential flaws in the corresponding OAuth clients.

To this end, we develop `OASIS`, a static analyzer capable of detecting COSF vulnerabilities in native apps. To our knowledge, existing OAuth testing tools [7], [8], [33], [34], [35], [36], [37] neither cover the use case of account linking in connector ecosystems nor detect session fixation flaws.

`OASIS` follows a two-phase approach, with the workflow overview shown in Fig. 14. Phase I filters Android applications that implement account linking, while Phase II detects COSF vulnerabilities without executing actual OAuth flows or attacks. Another major technical challenge lies in accommodating diverse Android app types, such as regular native apps written in Java or Kotlin, hybrid apps developed with Ionic or Capacitor, and cross-platform apps built with React Native. `OASIS` therefore adopts a lightweight design to ensure broad compatibility across such variants.

6.2. Phase I: Account Linking Identification

The first phase extracts Android apps that likely support OAuth-based account linking from a large corpus. We utilize AndroZoo [38], a large-scale research dataset of Android packages (APKs), and apply a multi-stage filtering pipeline:

Data Preprocessing. We begin with apps in the *Productivity* category, which, based on our preliminary study, exhibit the highest prevalence of account linking. To ensure relevance, we include only actively maintained apps (updated within the last 6 months) with at least 50K downloads. Each APK is decompiled using `Jadx` [39] to inspect source code and resource files. JavaScript bundles are unpacked into modules, and Hermes bytecode from React Native apps is decompiled using `hbc-decompiler` [40].

Heuristic-based Filtering. We then apply a set of static heuristics to exclude irrelevant apps and isolate OAuth confidential client implementations from the majority use case of public clients:

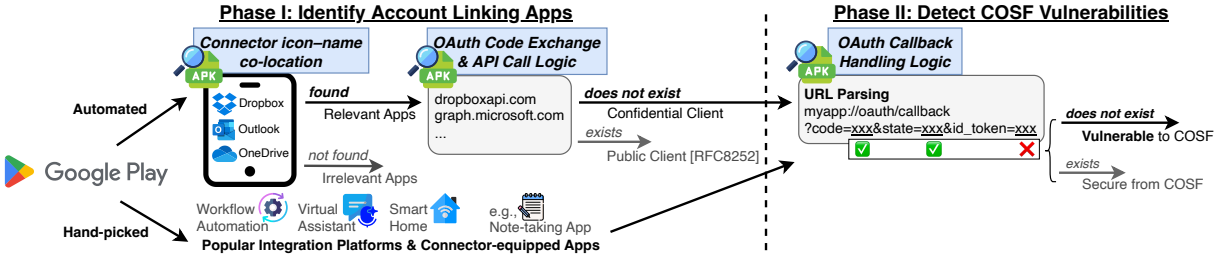

Figure 14. Overview of OASIS (OAuth Session Integrity Scanner). It follows a two-phase static approach to (I) identify native apps with account linking support, and (II) detect COSF vulnerabilities.

- **Connector Icon–Name Co-location.** We detect user interfaces (UIs) indicative of account linking by searching resource files for hard-coded connector brand names (*e.g.*, "Dropbox", "OneDrive") and corresponding logos (based on icon filenames and RGB values of vector graphics). The co-location of references to the connector's name and logo within the same class or module constitutes a typical account linking interface.
- **Rule Out OAuth Public Clients.** In Android, most OAuth use cases involve accessing resources directly from the device as public clients (first-order OAuth). In contrast, native apps in connector ecosystems operate as confidential clients (second-order OAuth), where resources are accessed by the app's cloud. We therefore exclude apps that directly invoke connectors' OAuth token endpoints or APIs on-device; the presence of such hard-coded endpoints indicates a public client implementation. Endpoints for major connectors are listed in Table 6 of Appendix B.
- **Exclude Local App Interactions.** As a supplementary filtering step, we remove apps that use intents [41] to launch locally installed native apps of the connectors (*e.g.*, Dropbox's file chooser [42]), since such interactions do not involve OAuth flows.

Scope of Connectors. We focus on 3 representative connectors: Dropbox, OneDrive, and Outlook Calendar. These connectors are well-suited for OASIS' workflow, because (1) they avoid the potential interference from OAuth-based Single Sign-On (SSO) mechanisms (Dropbox is not an SSO provider, and Microsoft SSO is rarely used [43]); and (2) they support silent authorization, enabling stealthy one-click attacks (see §4.1.2). Due to these considerations, Google products (*e.g.,* Google Drive and Calendar)[5] are not in scope.

Automated and Manual Collection. We augment automated filtering with a manually curated application list to capture cases missed by static heuristics. Key sources include integration platforms reported in prior research [1], along with popular connector-equipped apps such as note-taking apps. Applications without Android versions are retained separately for evaluation in §7. This process yields a refined set of apps with verified account linking functionality, which are subsequently analyzed in Phase II.

5. Google's AS does not issue refresh tokens without explicit user consent (according to [44, §11]), which incidentally mitigates one-click COSF attacks on many OAuth clients that require refresh tokens to be returned.

6.3. Phase II: COSF Vulnerability Detection

In Phase II, we analyze how each app handles the OAuth callback. Our key insight is that native apps that robustly defend against COSF by returning to the original UA (§4.3) would process the OAuth callback within their local codebase, making them identifiable through static analysis.

This phase extends the exclusion-based approach by inferring potential vulnerabilities from the *absence* of OAuth callback-handling patterns. Specifically:

- If an app contains a class or module that parses both the `code` and `state` parameters from the OAuth callback URI, but not the `id_token`, we classify it as **Secure**.
- Otherwise, the app is marked as **Vulnerable**.

This heuristic rests on two observations: (1) Mandating the joint presence of `code` and `state` helps exclude OAuth-unrelated logic that happens to use either parameter name. (2) Excluding the `id_token` parameter avoids interference from OpenID Connect [44], an OAuth-based SSO protocol orthogonal to account linking in connector ecosystems.

Implementation. Regardless of the app development framework, handling OAuth callbacks on Android ultimately relies on native processing of deep links or app links [45] via intents in Java (or Kotlin). In most cases, the intent URI is also parsed in Java (or Kotlin), as illustrated in Listing 1, before being passed to other frameworks or languages for further processing.

```java
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    // Handle OAuth callback
    Uri uri = getIntent().getData();
    if (uri != null) {
        // Parse callback URI,
        // e.g., myapp://oauth/callback?code=12345&state=xyz
        String code = uri.getQueryParameter("code");
        String state = uri.getQueryParameter("state");
        // Further Processing
        ...
    }
}
```

Listing 1. Example of OAuth callback handling logic in Android

Therefore, for each app selected in Phase I, we locate the callback-parsing logic in the decompiled source code. To improve precision, we also account for corner cases where URL parsing is implemented in JavaScript, which can occur in hybrid or cross-platform apps. Specifically, we employ a regular expression-based approach to identify method names

(*e.g.,* `getQueryParameter()`) and arguments (*e.g.,* `"code"`) commonly used in OAuth callback parsing. This simple but effective method maximizes multi-language compatibility. Implementation details are summarized in Table 7.

6.4. Discussions

Testing Scope. While `OASIS` is implemented to detect COSF vulnerabilities in Android apps, the methodology generalizes to iOS and desktop apps. Moreover, a vendor likely exhibits the same vulnerability across all native app offerings, as COSF is an OAuth client-specific backend flaw.
Threats to Validity. We acknowledge several limitations of our approach: (1) In Phase I, native apps that fetch the account linking page dynamically from the backend elude the static heuristics. Typical examples include integration platforms that host hundreds of connectors, where the connector names and icons are unlikely to be hard-coded in the app's code. This limitation is partly mitigated by the manual collection, where such platforms are explicitly selected. (2) In Phase II, native apps that ensure session integrity by re-authenticating end-users in the external UA (defense D6) do not exhibit detectable callback patterns, potentially causing false positives. (3) Native apps that handle OAuth callbacks locally (thus appearing secure) can still be vulnerable if their backends omit session integrity checks. Such cases cannot be captured by the Phase II analysis, resulting in false negatives. (4) Native app obfuscation: As `OASIS` identifies resource files, method invocations, and web API calls—rather than control or data flows—using OAuth domain knowledge, it is relatively robust to obfuscation.

We argue that these are inherent limitations of a static analysis approach. Dynamic analysis, however, is impractical at scale because it requires substantial manual effort to locate account linking interfaces and trigger OAuth flows for Phase I, which vary widely across app UIs and are often gated behind subscription paywalls. For Phase II, a dynamic approach may also suffer from degraded accuracy due to heterogeneous implementations, while offering limited practical benefits over manual attack reproduction.

We acknowledge that Phase I approach is non-exhaustive and represents a lower bound on real-world account linking support. Nevertheless, since account linking is far less prevalent than general OAuth or SSO adoption, our analysis at this scale still provides a reasonable characterization of the current landscape (see §7). For Phase II, the limited connector choices do not bias vulnerability detection accuracy, as identified OAuth callback logic is connector-agnostic.

7. Evaluation

This section presents our measurement and case studies.

7.1. Experimental Setup

Initial Triage. The cut-off date for our data collection from AndroZoo [38], [46] was August 2025. Following `OASIS` Phase I (§6.2), we collected ***38 connector-equipped apps and integration platforms***, including 12 obtained through automated filtering and 26 hand-picked. We followed §3.3 to analyze their architectural patterns. For vendors offering Android mobile apps (27 in total), we followed `OASIS` Phase II (§6.3) to automatically detect COSF vulnerabilities. We further surveyed agentic AI infrastructures across the industry using keywords such as "Agent Auth", "Agent Builder", and "Agentic Platform", identifying ***8 OaaS providers***. Cross-tenant threats were triaged via manual inspection of each vendor's developer documentation and runtime authorization URLs. Note that although the `COVScan` tool proposed in [1] can be adapted to detect Cross-tenant COAT, it was unnecessary, as all tested vendors (see Table 5) use a shared `redirect_uri`, causing early exit in `COVScan`'s detection flow [1, §7.1].
Manual Verification. For all **46** vendors, we then manually conducted proof-of-concept (PoC) exploits to verify the triaged vulnerabilities, following the attack descriptions in §4 and §5. In the testbed setup, we isolated the attacker and victim using two browser profiles (for web apps) or two devices (for native apps). We used separate accounts for test users (of apps and connectors) as well as test tenants, ensuring no shared state or information exchange between the attacker and victim except for the attack URL.
Human Effort. In the evaluation, initial testing took around 15 minutes per vendor. Producing detailed PoCs and vulnerability reports for responsible disclosure took about 2 hours per regular vendor and 1–2 days per OaaS vendor.

7.2. Measurement Study

Among the 46 vendors surveyed, 42 were found susceptible to account mislinking.

7.2.1. Cross-user Attacks.

Evaluation Results. Table 3 shows a breakdown of cross-user vulnerabilities: 40 vendors are affected by COSF, of which 3 and 4 are also vulnerable to App Account Login CSRF and Connector Account Forced Linking, respectively. For COSF alone, 16 vendors are vulnerable to the case where a standalone OAuth session ID is introduced, while the remaining 24 transfer the OAuth session via the `state` parameter. Four apps require advanced techniques for practical exploitation (labeled as Adv in Table 3), such as bypassing short-lived pre-authorization request URL timeouts, or sending attacker-controlled OAuth completion requests after the callback (see Appendix A.2–Ineffective Defenses).
RFC Compliance of `state` Matching. Among the 16 vendors introducing a separate OAuth session ID, only 3 fail to bind the `state` parameter to the OAuth session in cookies (which is a form of UA session). This suggests that most developers strive to meet RFC requirements [28, §4.7.1] of `state`–*UA session* binding despite cross-origin or cross-UA challenges. However, vulnerabilities remain if the OAuth session itself is not securely bound to the user session.
Cross-Origin and Cross-UA Deployments. We identified 27 cross-origin and 38 cross-UA vendors. To characterize vendors' architectural patterns, we first consider *origin*

Table 3. EVALUATION OF CROSS-USER ATTACKS IN CONNECTOR ECOSYSTEMS: 40 VENDORS BREAK OAUTH SESSION INTEGRITY.

Category	Vendor Type	Vendor Name	Collection	#DLs	Architectural Pattern §3.3		OAuth Session Fixation (COSF) §4.1			App Account Login CSRF §4.2.1	Connector Account Forced Linking §4.2.2
					Cross-Origin /Cross-Site	Cross-UA	Standalone sessionID	Session in `state`	Detection		
Connector-Equipped Apps	Profile Connections	A Top Social Platform	M	500M+	○	■		⚠	TP		
	Calendar Sync	A Top Note-Taking App	M	100M+	◑	■		⚠	TP		
		Akiflow	M	50K+	◑	■	⚠		TP		
		149 Live Calendar	A	500K+	○	■	⚠		TP		
		Cupla	M	100K+	◑	■		⚠ *Adv*	TP		
		FlowSavvy	A	50K+	○	■	*Secure*		TN		
		Habitify	A	500K+	●	■		⚠	TP		
		Smart Noter	A	1M+	○	■		⚠	TP		
		Upmeet	A	100K+	○	■		⚠	TP		
		Setmore	A	500K+	◑	■		⚠	TP		
		Harvest	M	100K+	◑	□	⚠		N/A	⚠	
		An Event Management Platform	M	N/A	●	□		⚠	N/A		
		Todoist	A	10M+	○	■	*Secure*		TN		
	File Sync / Data Source Import	OneDrive	A	5B+	●	■		⚠	FN		
		Speechify	A	10M+	◑	■		⚠	TP		
		JotterPad	M	5M+	○	■	⚠		TP		
		Tanka	M	10K+	○	■		⚠	TP		
		Mylio Photos	A	100K+	◑	■		⚠ *Adv*	TP		
		Nozbe Classic	A	100K+	◑	■	⚠		TP		
		Cloze	A	100K+	○	■	⚠		TP	⚠	
		A Top Note-Taking App	M	10M+	○	■	⚠		TP		
Integration Platforms	Workflow Automation Platforms	IFTTT	M	10M+	○	■	⚠		TP	⚠	
		Slack (Workflows)	M	10M+	◑	■	⚠ *Adv*		N/A		
		A Top Productivity Platform	M	1M+	●	■	⚠		N/A		
		n8n	M	N/A	○	□		⚠	N/A		
		A Top AI Orchestration Platform	M	N/A	◑	□	⚠ *Adv*		N/A		
		Integrately	M	N/A	○	□	⚠		N/A		⚠
		Pipedream	M	N/A	○	□		⚠	N/A		
	Virtual Assistants	Amazon Alexa	M	100M+	◑	■	⚠		TP		
		Baidu Xiaodu	M	5M+	○	■		⚠	TP		⚠
		Alibaba AliGenie	M	5M+	●	■	⚠		FN		⚠
		Le Chat by Mistral AI	M	1M+	◑	■		⚠	TP		
		Perplexity AI	M	10M+	○	□		⚠	N/A		
		Manus	M	1M+	○	□		⚠	N/A		⚠
		ChatGPT	M	1B+	○	■	*Secure*		TN		
		Claude	M	10M+	○	■	*Secure*		TN		
	Smart Homes	Samsung SmartThings	M	1B+	◑	■		⚠	FN		
		Xiaomi Home	M	50M+	●	■	⚠		TP		
Agentic AI Infrastructures	Token Vault / Agent Auth Solution / AI Agent Builder / AI Gateway (OAuth-as-a-Service)	Amazon AgentCore	M	N/A	●∀	■∀	⚠		N/A		
		Azure API Management	M	N/A	●∀	■∀	*Secure*		N/A		
		Microsoft Copilot Studio	M	N/A	●∀	■∀	*Secure*		N/A		
		Composio	M	N/A	●∀	■∀		⚠	N/A		
		Arcade	M	N/A	●∀	■∀		⚠	N/A		
		ByteDance Coze	M	N/A	●∀	■∀		⚠	N/A		
		Nango	M	N/A	●∀	■∀		⚠	N/A		
		ACI.dev	M	N/A	●∀	■∀		⚠	N/A		
Total		46	—	—	27/46	38/46	**16/46**	+ **24/46**	—	3/46	4/46

🌐 / 📱: Web/Native app *available*, and with *Account Linking support* (Desktop app 🖥 as fallback if Mobile app 📱 is unavailable or unsupported).
○: Same-Origin (and Same-Site); ◑: Cross-Origin but Same-Site; ●: Cross-Site. □: Same-UA; ■: Cross-UA. ⚠: Vulnerable. *Adv*: Advanced Attack Techniques Required.
Vendor Collection: A = (semi-)Automated, M = Manual. Vendor collected following `OASIS` Phase I (§6.2).
#DLs: Number of Downloads, statistics from Google Play (or Tencent Yingyongbao for Chinese-only apps).
∀: Support Heterogeneous Publish Channels (within any origin and/or UA, including deployed as bots in Instant Messaging software), beyond the vendor's official Web app or Native app.

as the primary security boundary, enforced by the same-origin policy. A cross-origin deployment typically implies that an app's main functionalities and its OAuth client are viewed, if not deployed, as separate components internally.

Cross-Site Deployments. We further use the *same-site* vs. *cross-site* distinction to capture whether the callback endpoint can access the app's session cookies. It is interesting to investigate why some vendors host callback endpoints cross-site, which *prevents* them from utilizing the user session. For instance, Habitify hosts its callback endpoint on Google Cloud Functions while keeping other functionalities under its own domain. A leading event management platform, although not an OaaS provider, hosts OAuth functionalities on a centralized server for templated event websites deployed on custom domains. Microsoft acquired Mover.io in 2019 to enhance OneDrive's file migration feature, yet its OAuth callback endpoint remained on a non-Microsoft domain.

Same-Origin and Same-UA Deployments. We observed 5 vendors that fall into neither cross-origin nor cross-UA categories. In these cases, secure OAuth practices should, in principle, be straightforward, reducing the problem to implementation flaws within the textbook OAuth model that involves a single origin and UA. Despite this, we speculate that developers may assume that once authorization context is encoded in the OAuth session, the callback endpoint no longer needs to consume the user session, resulting in a false sense of security. This underscores the importance of awareness of COSF even in textbook OAuth deployments. Another possibility is that the OAuth client, despite same-origin, is implemented as a standalone component separate from those handling user sessions; in such cases, the cross-origin countermeasure described in D1 and D3 still apply.

Vulnerability Detection with `OASIS`.

Phase I. The size of initial dataset is 3.6 million non-game apps from AndroZoo, landing 61,087 apps after filtering by update date and number of downloads. Upon preprocessing, 3,143 APKs of productivity apps were analyzed. As shown in Table 4, 26 Android apps were collected in Phase I. We

Table 4. EVALUATION OF OASIS FOR COSF VULNERABILITY DETECTION IN MOBILE APPS

Phase I: Collection (§6.2)			Phase II:† Detection (§6.3)			
Method	# Selected	# Excluded	# TP	# FN	# TN	# FP
(semi-)Automated	12	14	20	3	4	0
Manual	15	N/A				

† Positive (P): vulnerable; Negative (N): secure.

then manually verified the presence of account linking and excluded 14 apps: 6 referenced dead code, 3 required paid subscriptions without free trials, 3 were proprietary apps, and 2 offered account linking only in web versions.

Phase II. The remaining 12 Android apps were confirmed to support account linking and analyzed along with 15 manually selected Android apps in Phase II. Of the 27 vendors, 20 were correctly identified as vulnerable (true positives), 4 were correctly identified as secure (true negatives), with 3 false negatives and no false positives. False negatives include Samsung SmartThings (absence of server-side session integrity check despite native callback handling, see §6.4), OneDrive, and Alibaba AliGenie (interference from other secure OAuth features). To further validate the effectiveness of our approach, we scanned older versions of the 4 true negative apps that predate their account linking support. Three were correctly classified as lacking OAuth callbacks, while Claude was misclassified due to non-standard SSO flow that did not use the `id_token` parameter.

7.2.2. OAuth-as-a-Service Providers. We evaluated cross-user and cross-tenant attack vectors in OaaS providers, with results summarized in Table 5.

COSF. Six of the eight providers were vulnerable to COSF. Their respective fixes at the time of writing are listed in the table. Notably, one vendor exposed open redirect issues due to improper COSF defenses (see Appendix A.2).

Cross-tenant Client ID Confusion. We examined how vendors support and promote shared client IDs across tenants. The four vulnerable vendors treat shared client IDs as part of their business model, making BYOC optional rather than a security requirement. For example, Arcade previously stated that "it *can* be useful" for tenants to configure their own client IDs, but now states: "if you are building a multi-user production app, you *must* obtain your own OAuth app credentials and add them to Arcade." [48] after our report.

Cross-tenant COAT. We identified 7 susceptible OaaS providers, all of which use a shared `redirect_uri` across tenants and connectors. As a result, any connector in any tenant's agent relying on these OaaS Token Vaults is vulnerable to Cross-tenant COAT.

As a concrete example, the `redirect_uri` in Amazon Bedrock AgentCore `https://bedrock-agentcore.<AWS_REGION>.amazonaws.com/identities/oauth2/callback` was shared between the malicious tenant's connector and benign tenant's connector (*e.g.,* Dropbox). An attacker can exploit this by tricking a victim into interacting with an attacker-controlled agent. During account linking, the malicious connector's authorization endpoint `https://attacker.com/authorize` issues a crafted redirect to `https://www.dropbox.com/oauth2/authorize`. This Dropbox authorization URL embeds query parameters collected from attacker's interaction with the benign agent's Dropbox connector (*i.e.,* with Dropbox's `client_id`, the shared `redirect_uri`, and a fresh PKCE `code_challenge`), while reusing the `state` value of the victim's ongoing interaction. Eventually, the attacker can redeem the stolen auth code by continuing their own OAuth flow with Dropbox (*i.e.,* the one associated with the `code_challenge`), thereby completing the account takeover (PoC in [15]). Notably, the attack does not violate CSRF or PKCE protections. Amazon resolved this vulnerability by assigning globally unique, per-connector `redirect_uris` (by appending a UUID suffix to the original URI) and enforcing matching in OAuth flows.

7.3. Case Studies

Slack (Workflow Builder). Slack enforced a 10-second timeout on pre-authorization request URLs, but this does not mitigate COSF. An attacker can bypass the limit by replaying the full OAuth initiation request that generates the pre-authorization request URL on their own server *after* the victim visits the attacker's website. The attacker then immediately redirects the victim to the fresh request URL, ensuring its validity in an attack.

Harvest. Harvest's COSF vulnerability affects only its web app, not its mobile app. On mobile, the app acts as a public client to obtain tokens and forwards them to the app backend for storage and API calls. While this avoids COSF, public clients are inherently less secure than confidential clients as they lack client authentication. Furthermore, access tokens intended for backend use are exposed on-device, which could have been avoided with a confidential client. This illustrates the lack of guidance for implementing secure OAuth confidential clients in native apps.

Microsoft Copilot Studio. A severe Cross-tenant COAT vulnerability was identified in Microsoft Copilot Studio. Each custom agent (called a *Copilot*) uses the "Bot Framework Token Service" as its Token Vault. Microsoft designed this Token Vault to support broader use cases, where agent developers can not only request tool access (*e.g.,* Dropbox) but also authenticate users across different identity providers (IdPs, *e.g.,* Microsoft Entra ID), thereby enabling fine-grained access control over who can access each agent. Consequently, a Cross-tenant COAT attacker can configure a custom agent with a malicious IdP to access arbitrary well-authenticated agents on the Internet under the victim's identity, gaining access to all their gated resources.

Model Context Protocol (MCP). MCP [49] is an emerging standard for AI agent tool-use, in which MCP clients (OAuth clients) of an MCP host (*e.g.,* Claude Desktop) send access tokens to MCP servers (APIs) to access protected resources. While MCP OAuth [50] is usually considered *first-order OAuth*, where the MCP client acts as an OAuth public client, COSF attacks can manifest in two MCP scenarios:

1) MCP clients of remote MCP hosts. In this case, the MCP client is an OAuth confidential client. Users maintain

Table 5. OAUTH ACCOUNT MISLINKING VULNERABILITIES ACROSS OAUTH-AS-A-SERVICE PROVIDERS

Vendor Name	Cross-user Attacks [§4]		Cross-tenant Attacks [§5]	
	OAuth Session Fixation (COSF) [§4.1]	Countermeasures [§4.3, Appendix A]	Client ID Confusion [§5.1]	COAT [§5.2]
Amazon Bedrock AgentCore	⚠ Fixed	Post-redirect Pattern (D3) [3]	Secure	⚠ Fixed
Azure API Management	Secure, except for ⚠ Microsoft Copilot[¶] (Fixed)	Post-redirect Pattern (D3) or Extra Consent (D5) [47] ⚠ Open Redirect×3 (Fixed) [Appendix A.2]	Secure	⚠ Fixed
Microsoft Copilot Studio	Secure	Post-redirect Pattern (D3) or Manual PIN (D4)	Secure	⚠ Fixed
Composio	⚠ Insufficient Fix	Shorter Timeout	⚠ "recommend" [§]	⚠[†]
Arcade	⚠ Fixed	Post-redirect Pattern (D3) [48]	⚠ Fixed ("can"→"must")	⚠ Fixed
ByteDance Coze	⚠ Fixed	Extra Consent (D5)	⚠ Fixing	⚠ Fixed
Nango	⚠		Secure	⚠[†]
ACI.dev	⚠		⚠ "can"	N/A[‡]

[¶] A Microsoft first-party tenant of Azure API Management. [§] Requirement level of BYOC (Bring Your Own `client_id`), the same below.
[†] Cross-tenant COAT feasible via pre-built connectors with customizable AS endpoints from attacker's tenant. [‡] No custom(izable) connector support for any tenant.

sessions between their UA and the remote MCP host, and tokens from MCP servers are linked to the MCP host's user identity, making this a form of second-order OAuth (*i.e.,* account linking). We found 2 MCP clients (Manus AI and Mistral AI's Le Chat; see Table 3) susceptible to this issue.

2) MCP servers with downstream API authorization. If an MCP server relies on downstream APIs whose credentials are also fetched via OAuth and associated with the MCP session (between MCP client and server), then the downstream token retrieval process effectively becomes account linking. This pattern is standardized in the MCP *URL elicitation* proposal [51] by Arcade, a vulnerable agentic AI vendor (see Table 5). In addition to reporting vulnerabilities in Arcade's OaaS product line [48], we discovered its MCP proposal was also vulnerable and informed Arcade accordingly. Arcade subsequently updated its MCP proposal to include a discussion of the session fixation attack vector [52].

8. Related Work

This section discusses related work from both academia and industry, and highlights how they differ from this work.

8.1. Related Academic Research

Connector vs. SSO Account Linking. In this work, the account linking concept generally aligns with [1]. It contrasts with SSO account linking [34], [37], [53], where a user's IdP account connects to an application's existing account system for SSO. In SSO account linking, account takeovers arise from typical OAuth CSRF vulnerabilities [2, §10.12], rather than from session fixation issues.

OAuth in Integration Platforms. While (Single-tenant) COAT attacks in integration platforms [1] are not the focus of this work, they constitute an entry in our attack framework for connector ecosystems (Table 1), and our Cross-tenant COAT attack in OaaS builds upon them. Regarding session integrity, session fixation was not distinguished from CSRF issues in [1]: the authors attributed the potential for "one-click" COAT attacks to missing CSRF protection [1, §4.3], but only analyzed the case of OAuth-initiation CSRF. They did not investigate other causes of CSRF-like behaviors, where the OAuth callback depends on the OAuth session rather than the pre-existing platform session to identify the user, thereby overlooking the threat modeling of session fixation. We address this gap by examining the motivations and implications of such designs from the perspective of OAuth architectural patterns, highlighting session fixation as a standalone attack vector across connector ecosystems.

Cross-Origin OAuth. DISTINCT [34] identified security issues in dual-window SSO flows, including those that cross origins. However, the study focuses exclusively on SSO and does not cover OAuth authorization scenarios. A key distinction also lies in the flow assumptions: SSO may rely on cross-origin in-browser communication (InBC) mechanisms to return authentication credentials to the application's primary origin, whereas our COSF attack arises precisely because such a return is absent. Consequently, cross-origin InBC mechanisms such as `postmessage` could serve as an alternative to HTTP redirects when enforcing the COSF defense of post-redirect pattern (D3) to return to the original authenticated origin.

Brokered OAuth. Like SSO Brokers [54], OaaS with Token Vaults also extend three-legged OAuth into a four-actor architecture. However, the two ecosystems exhibit distinct security issues. The fundamental difference is that brokered SSO designs chain two SSO flows side by side (App↔Broker and Broker↔AS), whereas only a single OAuth flow is involved in OAuth-connection-based OaaS designs. In the latter, the Token Vault acts as the sole OAuth client; applications retrieve access tokens by 1) *authenticating to* the Token Vault (§3.3), and 2) presenting *OAuth connection IDs* (§3.4), which are non-secret identifiers. Consequently, each application is exempt from the burden of implementing OAuth client logic or managing tokens.

8.2. Related Discussions in Industry

Despite active discussions on OAuth in the industry, particularly within standardization bodies such as the IETF OAuth Working Group and OpenID Foundation, the specific issues uncovered in this work were previously unaddressed.

OAuth Session Integrity. Session integrity issues are a classic threat in web security. Prior research has extensively analyzed cookie integrity, leading to multiple updates to the cookie standard [55], [56]. While attack techniques such as *cookie tossing* are also feasible in OAuth [57], [58], such issues depend on ad-hoc OAuth implementations and fall outside the scope of this work.

Existing OAuth research on session integrity primarily focuses on (Login) CSRF [2], [59], [60], [54]. Regarding session fixation, one such attack was previously identified in OAuth 1.0 [61], where the "request token" functioned as an OAuth session and could be fixated on a victim's device. In OAuth 2.0, the concept of an OAuth session is not explicitly defined in the protocol specification, leaving only the `state` parameter which may carry authorization state information. However, as our study reveals, connector ecosystems may inevitably introduce standalone OAuth sessions through `state` parameters or standalone session IDs, and, more importantly, expose an *anti-pattern* in which clients solely rely on the fixated OAuth session to identify the current authorization context, thereby associating the access token with an unintended user at the client.

The OAuth community has also discussed remote phishing in cross-device OAuth flows [62, §5.4], [63], as well as session fixation in OpenID for Verifiable Presentations (OpenID4VP) [64, §14.2]. However, such ecosystems assume significantly modified versions of the OAuth protocol, handling authorization between a consumption device and an authorization device, or presentations between a verifier and a wallet, with attacks and defenses applicable only within those contexts. In contrast, we are the first to analyze session fixation patterns in the standard authorization code grant in same-device OAuth flows, which remains susceptible to practical real-world attacks.

Furthermore, our COSF attack is *stealthier*: previous cases required significant social engineering, as victims *must* perform user interaction (such as entering a user code) in cross-device flows [62, §3.3], and *should* provide explicit consent in OpenID4VP [64, §15.1]. In contrast, COSF attacks can be triggered simply by tricking victims into clicking a hyperlink, and attackers can leverage silent authorization configured by mainstream ASes to bypass explicit user consent, resulting in one-click attacks. Finally, to our knowledge, no prior work has conducted measurement studies or proposed vulnerability detection methods for session fixation attacks in OAuth(-derived) protocols.

Decoupled OAuth Architectural Patterns. The "OAuth for Browser-Based Applications" specification [65] describes the Backend-for-Frontend (BFF) and Token-Mediating Backend (TMB) OAuth architectural patterns. Although account linking in native apps resembles BFF, and OaaS providers that return raw access tokens to the apps are comparable to TMB, there are two key differences:

(1) According to [65], the retrieved tokens are tied to the BFF or TMB's own session, which are generated by BFF or TMB *after token retrieval*. BFF or TMB-managed tokens are not linked to the user's identity at the browser-based app. By contrast, the account linking scenario associates tokens with the app's user identity, as indicated by the app's *pre-existing user session*. (2) The specification's security considerations are scoped to a strong threat model of *compromised* applications (exploited by malicious JavaScript) in *browser-based apps*, whereas the COSF attack in this work assumes malicious (yet unprivileged) users of a *benign* application in *web and native apps*.

9. Conclusion

This work presents the first systematic study of architectural patterns underlying OAuth-based account linking in connector ecosystems. We develop a unified security framework for *account mislinking* threats, including Cross-user OAuth Session Fixation (COSF), the first session fixation attack against implementations of the standard OAuth 2.0 authorization code grant flow. Our semi-automated measurement uncovers 40 real-world apps, platforms, or infrastructures vulnerable to COSF, and further reveals Cross-tenant confused deputy flaws in 8 OAuth-as-a-Service providers.

Our analysis exposes fundamental gaps that warrant immediate attention from both the research and standards communities: (1) Existing OAuth specifications lack normative guidance on maintaining session integrity in cross-origin, cross-user-agent, and separate-responsibility scenarios. (2) Emerging OAuth-as-a-Service architectures reshape traditional trust boundaries. While they offer convenience for agentic AI development, their multitenancy may inadvertently expose end-users to unauthorized access. This work represents an initial step toward addressing these gaps; while the proposed defenses are intuitive and empirically motivated, formal security proofs remain important future work.

We are working with affected vendors and the IETF to harden implementations and advance security best practices.

Ethics Considerations

Considerations for Human Subjects Research. All experiments were conducted on self-registered test accounts, without putting humans at risk.

Considerations for Vulnerability Disclosure. In accordance with responsible disclosure practices, we reported our findings to all affected vendors. For each contacted vendor, we provided a bug report with detailed reproduction steps and mitigation strategies, along with a PoC video. As of this writing, we have received acknowledgments from 20 vendors (over 15 of which have fixed their issues) and $35,850 in bug bounties from 11 vendors. We also participated in meetings with engineering teams at Microsoft, Arcade, and ByteDance. Arcade published a blog detailing their fix and system re-architecture as a community call-to-action, describing our disclosure as "the first major identity vulnerability in agentic AI" [66]. ByteDance issued a formal letter of appreciation. Amazon [3] and Arcade [48] reused our root-cause analysis in their updated documentation.

We re-tested the vulnerabilities following our disclosure. Among vendors whose systems were either fixed or not vulnerable, 18 vendors adopted COSF defenses D1–D3, 1 adopted D4, 3 adopted D5, and 2 adopted D6. Some vendors find deploying mitigations challenging, as robust fixes may require cooperation from *end-users* (to upgrade from deprecated vulnerable native app versions) or *OaaS tenant developers* (to update their agents and/or connectors).

We engaged with the IETF OAuth Working Group, authoring an Internet-Draft [16] to propose updates to the published OAuth Security Best Current Practice document

(RFC9700 [28]), and subsequently presented and discussed the issues during Interim and IETF meetings [67] with the Working Group members. The draft highlights (1) the application scenario of OAuth Connections/Account Linking, and, for both COSF and COAT, (2) the attack descriptions and (3) proposed countermeasures, especially cross-origin/UA considerations for COSF and cross-tenant considerations of globally-unique identifiers for COAT. Our Internet-Draft has been adopted as an OAuth Working-Group Draft, reflecting recognition of its practical relevance and impact.

LLM Usage Considerations

LLMs were used for editorial purposes in this manuscript, and all outputs were inspected by the authors to ensure accuracy and originality.

Acknowledgments

We thank the anonymous reviewers for their helpful feedback and our shepherd for the constructive comments and guidance. This work is supported in part by the CUHK MobiTeC Fund (Project No. 6901539) and the CUHK Strategic Impact Enhancement Fund (Project Nos. 399857576 and 456489500).

References

[1] K. Luo, X. Wang, P. H. A. Fung, W. C. Lau, and J. Lecomte, "Universal cross-app attacks: Exploiting and securing OAuth 2.0 in integration platforms," in *34th USENIX Security Symposium (USENIX Security 25)*, 2025, pp. 3221–3238.

[2] D. Hardt, "The OAuth 2.0 Authorization Framework," RFC 6749, Oct. 2012.

[3] AWS Documentation, "OAuth 2.0 authorization URL session binding - Amazon Bedrock AgentCore," https://docs.aws.amazon.com/bedrock-agentcore/latest/devguide/oauth2-authorization-url-session-binding.html.

[4] Microsoft Learn, "About Credential Manager in Azure API Management," https://learn.microsoft.com/en-us/azure/api-management/credentials-overview.

[5] D. Fett, R. Küsters, and G. Schmitz, "A comprehensive formal security analysis of OAuth 2.0," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 1204–1215.

[6] D. Fett, R. Küsters, and G. Schmitz, "The web SSO standard OpenID Connect: In-depth formal security analysis and security guidelines," in *2017 IEEE 30th Computer Security Foundations Symposium (CSF)*, 2017, pp. 189–202.

[7] T. A. Rahat, Y. Feng, and Y. Tian, "OAUTHLINT: An empirical study on OAuth bugs in Android applications," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2019, pp. 293–304.

[8] ——, "Cerberus: Query-driven scalable vulnerability detection in OAuth service provider implementations," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022, p. 2459–2473.

[9] W. Denniss and J. Bradley, "OAuth 2.0 for Native Apps," RFC 8252, Oct. 2017.

[10] CWE - Common Weakness Enumeration, "CWE-384: Session fixation," https://cwe.mitre.org/data/definitions/384.html.

[11] M. Kolsek, "Session fixation vulnerability in web-based applications," *ACROS Security*, 2002, http://www.acrossecurity.com/papers/sessionfixation.pdf.

[12] M. Johns, B. Braun, M. Schrank, and J. Posegga, "Reliable protection against session fixation attacks," in *Proceedings of the 2011 ACM Symposium on Applied Computing*, 2011, pp. 1531–1537.

[13] CWE - Common Weakness Enumeration, "CWE-441: Unintended proxy or intermediary ('confused deputy')," https://cwe.mitre.org/data/definitions/441.html.

[14] n8n-io/n8n, "fix(core): Improve the security on oauth callback endpoints," https://github.com/n8n-io/n8n/pull/11593.

[15] "Artifacts," https://mobitec.ie.cuhk.edu.hk/connector-oauth-security.

[16] T. Würtele, P. Hosseyni, K. Luo, and A. Fung, "Updates to OAuth 2.0 Security Best Current Practice," Internet Engineering Task Force, Internet-Draft draft-ietf-oauth-security-topics-update-01, Mar. 2026, work in Progress. https://datatracker.ietf.org/doc/draft-ietf-oauth-security-topics-update/01/.

[17] T. Lodderstedt, M. McGloin, and P. Hunt, "OAuth 2.0 Threat Model and Security Considerations," RFC 6819, Jan. 2013.

[18] IBM, "What is application integration?" https://www.ibm.com/think/topics/application-integration.

[19] AWS, "What is application integration? - app integration explained," https://aws.amazon.com/what-is/application-integration/.

[20] ——, "What is agentic ai? - agentic ai explained," https://aws.amazon.com/what-is/agentic-ai/.

[21] MDN Web Docs, "Same-origin policy - security," https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy.

[22] ——, "eTLD - glossary," https://developer.mozilla.org/en-US/docs/Glossary/eTLD.

[23] Arcade Docs, "Getting your API key," https://docs.arcade.dev/en/home/api-keys.

[24] AWS Documentation, "Get workload access token - Amazon Bedrock AgentCore," https://docs.aws.amazon.com/bedrock-agentcore/latest/devguide/get-workload-access-token.html.

[25] T. South, S. Nagabhushanaradhya, A. Dissanayaka, S. Cecchetti, G. Fletcher, V. Lu, A. Pietropaolo, D. H. Saxe, J. Lombardo, A. M. Shivalingaiah *et al.*, "Identity management for agentic AI: The new frontier of authorization, authentication, and security for an AI agent world," *arXiv preprint arXiv:2510.25819*, 2025.

[26] MDN Web Docs, "Set-Cookie header - HTTP," https://developer.mozilla.org/en-US/docs/Web/HTTP/Reference/Headers/Set-Cookie.

[27] D. Akhawe, A. Barth, P. E. Lam, J. Mitchell, and D. Song, "Towards a formal foundation of web security," in *2010 23rd IEEE Computer Security Foundations Symposium*. IEEE, 2010, pp. 290–304.

[28] T. Lodderstedt, J. Bradley, A. Labunets, and D. Fett, "Best Current Practice for OAuth 2.0 Security," RFC 9700, Jan. 2025.

[29] Microsoft Learn, "Protect against consent phishing - Microsoft Entra ID," https://learn.microsoft.com/en-us/entra/identity/enterprise-apps/protect-against-consent-phishing.

[30] P. Kasselman, D. Fett, and F. Skokan, "Cross-Device Flows: Security Best Current Practice," Internet Engineering Task Force, Internet-Draft draft-ietf-oauth-cross-device-security-12, Sep. 2025, work in Progress.

[31] N. Sakimura, J. Bradley, and N. Agarwal, "Proof Key for Code Exchange by OAuth Public Clients," RFC 7636, Sep. 2015.

[32] CWE - Common Weakness Enumeration, "CWE-601: URL redirection to untrusted site ('open redirect')," https://cwe.mitre.org/data/definitions/601.html.

[33] T. Innocenti, M. Golinelli, K. Onarlioglu, A. Mirheidari, B. Crispo, and E. Kirda, "OAuth 2.0 Redirect URI validation falls short, literally," in *Proceedings of the 39th Annual Computer Security Applications Conference*, 2023, pp. 256–267.

[34] L. Jannett, V. Mladenov, C. Mainka, and J. 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*, 2022, p. 1553–1567.

[35] S. Shi, X. Wang, and W. C. Lau, "MoSSOT: An automated blackbox tester for Single Sign-On vulnerabilities in mobile applications," in *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*, 2019, p. 269–282.

[36] P. Philippaerts, D. Preuveneers, and W. 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*, 2022, pp. 460–481.

[37] A. Bisegna, M. Bitussi, R. Carbone, L. Compagna, S. Ranise, and A. Sudhodanan, "CSRFing the SSO waves: Security testing of SSO-based account linking process," in *2024 IEEE 9th European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2024, pp. 139–154.
[38] K. Allix, T. F. Bissyandé, J. Klein, and Y. L. Traon, "AndroZoo: Collecting millions of Android apps for the research community," in *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*, 2016, pp. 468–471.
[39] skylot/jadx, "Dex to Java decompiler," https://github.com/skylot/jadx.
[40] P1sec/hermes-dec, "A reverse engineering tool for decompiling and disassembling the React Native Hermes bytecode," https://github.com/P1sec/hermes-dec.
[41] Android Developers, "Intent," https://developer.android.com/reference/android/content/Intent.
[42] Dropbox Developers Documentation, "Chooser - developers - Dropbox," https://www.dropbox.com/developers/chooser#android.
[43] L. Jannett, M. Westers, T. Wich, C. Mainka, A. Mayer, and V. 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)*, 2024, pp. 173–192.
[44] N. Sakimura, J. Bradley, M. Jones, B. De Medeiros, and C. Mortimore, "OpenID Connect Core 1.0 incorporating errata set 2," *The OpenID Foundation, Specification*, 2023, https://openid.net/specs/openid-connect-core-1_0.html.
[45] Android Developers, "About deep links | App architecture," https://developer.android.com/training/app-links.
[46] M. Alecci, P. J. R. Jiménez, K. Allix, T. F. Bissyandé, and J. Klein, "AndroZoo: A retrospective with a glimpse into the future," in *Proceedings of the 21st International Conference on Mining Software Repositories*, 2024, pp. 389–393.
[47] Azure/azure-tokens, "Phishing attack vulnerability," https://github.com/Azure/azure-tokens/blob/master/docs/phishing-attack-vulnerability.md.
[48] Arcade Docs, "Secure auth in production," https://docs.arcade.dev/en/home/auth/secure-auth-production.
[49] Anthropic, "Specification - Model Context Protocol," https://modelcontextprotocol.io/specification/2025-11-25.
[50] ——, "Authorization - Model Context Protocol," https://modelcontextprotocol.io/specification/2025-11-25/basic/authorization.
[51] N. Barbettini and W. Dawson, "SEP-1036: URL mode elicitation for secure out-of-band interactions," https://github.com/modelcontextprotocol/modelcontextprotocol/pull/887.
[52] Anthropic, "Elicitation - Model Context Protocol," https://modelcontextprotocol.io/specification/2025-11-25/client/elicitation#phishing.
[53] Web Security Academy, "Lab: Forced OAuth profile linking," https://portswigger.net/web-security/oauth/lab-oauth-forced-oauth-profile-linking.
[54] T. Innocenti, L. Jannett, C. Mainka, V. Mladenov, and E. Kirda, ""only as strong as the weakest link": On the security of brokered Single Sign-On on the web," in *2025 IEEE Symposium on Security and Privacy (SP)*, 2025, pp. 1009–1027.
[55] M. Squarcina, P. Adão, L. Veronese, and M. Maffei, "Cookie crumbles: breaking and fixing web session integrity," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 5539–5556.
[56] A. Bortz, A. Barth, and A. Czeskis, "Origin cookies: Session integrity for web applications," *Web 2.0 Security and Privacy (W2SP)*, 2011.
[57] Snyk Labs, "Hijacking OAUTH flows via cookie tossing," https://labs.snyk.io/resources/hijacking-oauth-flows-via-cookie-tossing/.
[58] Harel Security Research, "Zoom session takeover - cookie tossing payloads, OAuth dirty dancing, browser permissions hijacking, and WAF abuse," https://nokline.github.io/bugbounty/2024/06/07/Zoom-ATO.html.
[59] T. Lodderstedt, J. Bradley, A. Labunets, and D. Fett, "Best Current Practice for OAuth 2.0 Security," RFC 9700, Jan. 2025.
[60] R. Yang, G. Li, W. C. Lau, K. Zhang, and P. Hu, "Model-based security testing: An empirical study on OAuth 2.0 implementations," in *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, 2016, p. 651–662.
[61] OAuth Community Site, "OAuth security advisory: 2009.1," https://oauth.net/advisories/2009-1/.
[62] W. Denniss, J. Bradley, M. B. Jones, and H. Tschofenig, "OAuth 2.0 Device Authorization Grant," RFC 8628, Aug. 2019.
[63] D. Fett, "Cross-device session fixation and how the DC API solves it," https://danielfett.de/2025/03/10/cross-device-session-fixation/.
[64] O. Terbu, T. Lodderstedt, K. Yasuda, D. Fett, and J. Heenan, "OpenID for Verifiable Presentations 1.0," *The OpenID Foundation, Specification*, 2025, https://openid.net/specs/openid-4-verifiable-presentations-1_0.html.
[65] A. Parecki, P. D. Ryck, and D. Waite, "OAuth 2.0 for Browser-Based Applications," Internet Engineering Task Force, Internet-Draft draft-ietf-oauth-browser-based-apps-25, Jul. 2025, work in Progress.
[66] N. Barbettini, "How Arcade proactively addressed the first major identity vulnerability in agentic AI," https://www.arcade.dev/blog/arcade-proactively-addressed-coat-vulnerability-in-agentic-ai/.
[67] IETF 125, "Updates to OAuth 2.0 Security Best Current Practice," Mar. 2026, https://datatracker.ietf.org/meeting/125/materials/slides-125-oauth-updates-to-oauth-20-security-best-current-practice-00.

Appendix A. Additional Details on COSF Defenses

A.1. Post-redirect Defense in OAuth-as-a-Service

OaaS not only increases the risk of COSF attacks but also introduces new challenges for their defense. In OaaS, the agent (application)'s user session and its OAuth client are operated by separate entities, thus the callback endpoint is unable to interpret the agent's user session. It is therefore essential to clearly delineate responsibilities between the agent and the OAuth client.

As illustrated in Fig. 15, the *post-redirect pattern* defense in OaaS further requires the agent to extract the user identifier and forward it, along with the received authorization data, to the OAuth client for consistency check. A common implementation is as follows:

1) The agent (application) sets up a post-callback endpoint;
2) The Token Vault (OAuth client) redirects the browser to this endpoint after the initial OAuth callback, attaching an OAuth session-bound *nonce* (explained below);
3) The agent extracts the *user ID* from the *user session*, submitting it along with the *nonce*;
4) Token Vault uses the *nonce* to validate *OAuth session–user ID* binding, then proceeds with code exchange.

Note that during post-redirection, unlike the mechanism described in defense D3 (§4.3), which transparently for-

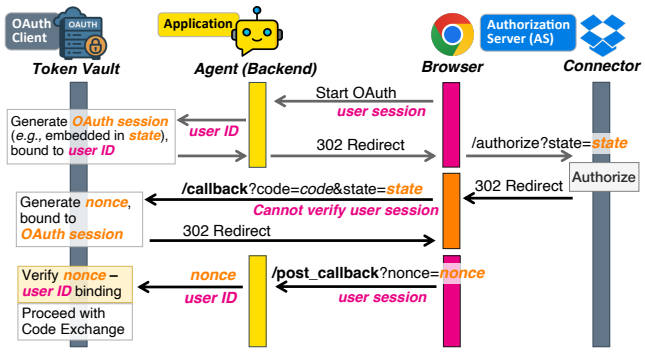

Figure 15. COSF Defense in OAuth-as-a-Service

wards the original authorization data (`code` and `state`) as a proxy, the OAuth callback may alternatively consume the authorization data and emit a fresh *nonce* for subsequent session integrity validation. Such design is common in OaaS, as it prevents applications from directly handling raw OAuth credentials, consistent with the decoupled business model. Two vendors that adopted this defense in their vulnerability fixes are documented in [3], [48].

Takeaway. Even in OAuth-as-a-Service architectures that aim to fully decouple OAuth client logic from the application, the application itself *must* still assume certain OAuth responsibilities for security purposes.

A.2. Common Pitfalls and Ineffective Defenses

The following discusses several general caveats: bad practices that should be avoided and invalid mechanisms that offer little value as a defense.

Common Pitfalls.

1) Do not set the app's full user session during the account linking process. Doing so enables Login CSRF attacks of app accounts (see §4.2.1).
2) Always generate OAuth session IDs dynamically with sufficient entropy. Predictable identifiers invite brute-force attacks (see §4.2.2).
3) Avoid treating the native app entirely as a public client per RFC8252 [9], *i.e.,* obtaining an access token on-device and then sending it to the app backend for account linking. This approach is susceptible to access token exposure (see the Harvest case study in §7.3).

Ineffective Defenses.

1) Proof Key for Code Exchange (PKCE) [31] does not resolve the problem. PKCE only preserves the binding between the authorization request and token request, but not between the initiation of OAuth (generating the authorization request) and the authorization request. In fact, in a COSF attack, both the authorization request and token request are completed in a single OAuth flow.
2) *Short-lived* (pre-)authorization request URL is ineffective as a sole defense (see the Slack case study in §7.3).
3) Additional "completion" steps after the OAuth callback are ineffective if they rely on attacker-controlled data. For example, sending an *OAuth completion request* with simply a success message (*e.g.,* `https://app.com/oauth/complete?success=true`) or reusing the OAuth session ID (*e.g.,* `https://app.com/oauth/complete?sessionID=foobar`) are insufficient. As such information is known to the attacker, the completion request can be forged and therefore cannot defend against COSF.

Caveat: Open Redirect. The post-redirect pattern (D3) is a robust defense against COSF. However, if an attacker can control the target location of the post-redirect, this would inadvertently create an auth code exfiltration channel, resulting in open redirect [32] flaws.

In our study, we discovered three real-world vulnerability instances in Azure API Management, an OaaS provider from Microsoft (see Table 5). While mitigating COSF via the post-redirect pattern defense, its Token Vault ("Credential Manager" [4]) exposed open redirect issues in Microsoft first-party applications that rely on it. In *Microsoft Copilot*, the Token Vault did not validate post-redirect URLs, allowing an attacker to tamper with them to leak authorization credentials to any attacker-controlled domain. In *Microsoft Power Apps* and *Azure Logic Apps*, the Token Vault validated post-redirect URLs only up to the root domain, leaving subdomains and path components as wildcards. As a result, an attacker could specify the post-redirect URL to locations capable of exfiltrating authorization credentials, for example, pages with Cross-site Scripting (XSS) or `postMessage` vulnerabilities [33], [34] on Microsoft-owned domains.

To prevent this, the post-redirect URL should be preconfigured at the OAuth client and validated using exact string matching in OAuth flows, without any wildcards. Ideally, the post-redirect URL should not accept any frontend-supplied values at all.

Insight. A common source of open redirects in textbook OAuth is the redirection from AS to the OAuth client [28, §4.11]: the auth code can be leaked if the OAuth callback location is controlled by an attacker. Here, the additional post-redirect from the OAuth client to the application can unintentionally reintroduce open redirect vulnerabilities if the COSF defense is not implemented securely.

Appendix B. Implementation Details of `OASIS`

The details of `OASIS` for identifying OAuth public client (in Phase I) and OAuth callback handling logic (in Phase II) are shown in Table 6 and Table 7, respectively.

Table 6. TOKEN AND API ENDPOINTS OF POPULAR CONNECTORS, REFERENCED BY `OASIS` PHASE I TO EXCLUDE OAUTH PUBLIC CLIENTS.

Connector	OAuth Token Endpoints	API Endpoints (Prefix)
Dropbox	`api.{dropbox,dropboxapi}.com /oauth2/token`	`{api,content}.dropboxapi.com`
OneDrive	`login.microsoftonline.com /common/oauth2/v2.0/token`	`graph.microsoft.com/v1.0 /{me/drive,drives/*}`
		`graph.microsoft.com/v1.0 /{groups,sites,users}/*/drive`
Outlook Calendar	`login.microsoftonline.com /common/oauth2/v2.0/token`	`graph.microsoft.com/v1.0 /me/{calendar*,events,findMeetingTimes}`

Table 7. SCREENING LOGIC USED BY `OASIS` PHASE II. AN ANDROID APP IS CONSIDERED TO HANDLE OAUTH CALLBACKS IF IT SATISFIES ANY OF THE FOLLOWING RULES. EACH RULE SPECIFIES PATTERNS THAT MUST AND MUST NOT BE PRESENT, EVALUATED WITHIN THE SAME CLASS OR MODULE.

Category	Detection Rule
Java (regular Native apps)	`Uri AND .getQueryParameter("code") AND .getQueryParameter("state") AND NOT .getQueryParameter("id_token")`
	`JSONObject AND .getString("code") AND .getString("state") AND NOT .getString("id_token")`
JavaScript / TypeScript (Hybrid apps or Cross-platform apps)	`(URL.searchParams OR URLSearchParams) AND .code AND .state AND NOT .id_token`
	`(URL.searchParams OR URLSearchParams) AND .get("code") AND .get("state") AND NOT .get("id_token")`

Appendix C. Meta-Review

The following meta-review was prepared by the program committee for the 2026 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

C.1. Summary

The paper examines the security of OAuth-based account linking in connector ecosystems and OAuth-as-a-Service (OaaS) infrastructures. It identifies new attack vectors including Cross-user OAuth Session Fixation (COSF) and cross-tenant confused deputy attacks, presents OASIS (a static analysis framework for detecting these vulnerabilities in Android apps), and evaluates the prevalence of the attacks across popular vendors, with responsible disclosure leading to vendor acknowledgments and fixes.

C.2. Scientific Contributions

- Creates a New Tool to Enable Future Science.
- Identifies an Impactful Vulnerability.
- Provides a Valuable Step Forward in an Established Field.

C.3. Reasons for Acceptance

1) **Identifies an Impactful Vulnerability.** The paper identifies a new attack surface for OAuth 2.0 in account linking scenarios that have not been previously studied. The findings involve major vendors including Slack and Amazon Bedrock AgentCore, with 40 vendors confirmed susceptible to COSF attacks and 8 OaaS providers susceptible to cross-tenant attacks. Most vendors acknowledged and partially deployed mitigations following responsible disclosure.
2) **Creates a New Tool to Enable Future Science.** The paper presents OASIS, a static analysis framework that enables the detection of account linking flaws and COSF vulnerabilities in mobile apps. This tool represents a first step toward automated analysis of OAuth-based account linking flows.
3) **Provides a Valuable Step Forward in an Established Field.** The paper advances understanding of OAuth security by analyzing the connector ecosystem/OaaS attack surface, combining empirical analysis, case studies, and recommended mitigations.

C.4. Noteworthy Concerns

1) **Limitations of the OASIS tool.** The analysis relies on static heuristics, works only on Android apps, and detects only COSF vulnerabilities. The effectiveness of the first phase of OASIS is questionable. The tool requires substantial manual effort to confirm findings.
2) **Mitigations.** The defenses presented are intuitive but not formally analyzed or proven to be robust. No systematic evaluation of mitigation effectiveness is provided, and it is unclear whether some attack vectors may remain even after proposed fixes are applied.