ChatGPT

OAuth Security Concerns and Implications for the Model Context Protocol (MCP)

Abstract

OAuth 2.0 is the de facto standard for delegated authorization on the web, but it was not designed with the fine-grained control and multi-system context of modern AI integrations in mind. This paper examines OAuth's security limitations – including coarse-grained permission scopes, an all-or-nothing user consent model, token theft vulnerabilities, and inconsistent implementations – and analyzes how these issues pose significant risks to the **Model Context Protocol (MCP)** used in AI systems. MCP enables AI models to interface with multiple tools and data sources, so any weakness in OAuth-based authentication can be amplified across systems. We provide a detailed overview of OAuth's known shortcomings (such as overly broad scopes and lack of selective authorization), then discuss how these weaknesses could lead to privilege escalation, cross-system exploits, and compliance challenges in an MCP deployment. Real-world incidents are presented to illustrate OAuth's vulnerabilities – from stolen GitHub OAuth tokens leading to source code breaches, to a Google OAuth phishing worm that spread via rogue permissions. The analysis highlights that without addressing OAuth's limitations, integrating AI models via MCP could expose organizations to severe security risks. We conclude by underscoring the urgency of recognizing and mitigating these OAuth-related issues in AI contexts, given their potential impact on sensitive, interconnected systems.

Introduction

OAuth 2.0 is widely used to grant applications limited access to user data on other services without sharing passwords. For example, logging in with Google or connecting a third-party app to a cloud service typically relies on OAuth's token-based authorization. While OAuth has enabled a rich ecosystem of integrations, its security model has known limitations that can be problematic in advanced use cases. One emerging use case is the **Model Context Protocol (MCP)** – a standard for connecting AI models (such as large language model assistants) to external data sources, tools, and services. MCP can be thought of as a "universal connector" that lets AI systems interface with multiple environments in a secure and standardized way [1]. In practice, an AI using MCP may need to access various platforms (e.g. code repositories, databases, cloud APIs) on a user's behalf. This raises the question: **How well do OAuth's security provisions hold up when an AI agent is granted broad OAuth tokens across numerous systems?**

This paper explores the security concerns of OAuth in depth and examines their impact on MCP-based AI integrations. The goal is to inform cybersecurity and technical professionals about the inherent risks – not to propose new solutions, but to highlight why existing weaknesses in OAuth could undermine the security of AI systems that rely on it. We begin by outlining OAuth's fundamental limitations in permission granularity, selective authorization, token security, and inconsistent implementation. Next, we analyze how these issues could translate into concrete threats in the MCP context, where an AI model's OAuth credentials to multiple services could be misused. We then discuss real-world cases that exemplify these vulnerabilities, such as breaches stemming from OAuth token theft and OAuth consent abuse. Finally, we conclude with reflections on why these risks are especially severe for AI-driven

systems using MCP, emphasizing the need for heightened awareness and mitigations when adopting OAuth in such contexts.

OAuth's Security Limitations

Coarse-Grained Permissions and Static Scopes

A well-known limitation of OAuth is its **coarse-grained permission model**. OAuth access tokens carry scopes that define what an application can do, but these scopes are often broad and inflexible. In many implementations, a single scope can grant sweeping access to a user's resources. For instance, GitHub's classic OAuth scope `repo` **"grants full access to read and write everything"** in all of a user's repositories [2]. This means if an application requests the `repo` scope, it gains complete control over all private and public repos of the user – far beyond any minimal necessity for a specific task. In fact, GitHub acknowledged that its legacy personal access tokens, which use coarse scopes, effectively had access to everything the user could access, and could last indefinitely [3]. Such broad scopes violate the principle of least privilege: an app may only need to read one repository or perform a single action, yet the token allows far more.

Compounding this issue is the static nature of OAuth scopes. Permissions are typically decided at the moment of user consent and remain unchanged for the lifetime of the token. There is no built-in mechanism for dynamically narrowing permissions once granted. If an application integrated via OAuth needs additional permissions later, it often must request a new token with expanded scopes, since tokens can't easily be modified or restricted on the fly. This static, **"all-or-nothing" consent** model forces users to either grant the full set of requested permissions or deny the integration entirely [4]. Users cannot selectively approve only certain permissions while withholding others – traditional OAuth consent screens do not offer granular checkboxes for each scope. (Notably, providers like Google are only recently moving toward *consent unbundling* to allow more fine-grained approval of scopes [4], highlighting that the earlier OAuth model was indeed all-or-nothing.)

The result of coarse, static scopes is an **over-privileged token** in many cases. Once an OAuth token is issued, it often includes more privileges than needed, and these privileges persist until the token is revoked or expires. Fine-grained authorization – e.g., granting access to only a specific file or a single function – is generally not natively supported by OAuth 2.0's scope design [5]. Attempts to use OAuth scopes for detailed authorization run into practical limits: defining extremely narrow scopes for every resource becomes impractical, and tokens (especially JWTs) can only carry so much scope information without hitting size limits [6] [7]. In summary, OAuth trades granularity for simplicity, which can leave integrations with far more access than is safe.

Lack of Selective or Contextual Authorization

OAuth's coarse scopes lead directly to a **lack of selective authorization** options for end-users. When a user is presented with an OAuth consent dialog, they must grant all requested permissions in bulk. There is typically no way to approve some permissions but not others. As a result, the user's control is binary – they authorize the application with a broad set of powers, or they cancel the process. This lack of nuance can be dangerous. A user may trust an app to do one thing but not another, yet OAuth doesn't let them finely distinguish those during consent. For example, an app might request access to both a user's Google Calendar and Gmail. The user could be comfortable sharing their Calendar but not their email. However, under a standard OAuth prompt, the user cannot selectively allow just the calendar scope; it's an **all-or-nothing consent screen** [4]. This pressure to over-consent can lead to users unknowingly granting more access than intended.

Moreover, OAuth lacks context-aware or selective activation of permissions. Once an app is authorized, every action it takes with the token is equally permitted. The protocol does not natively account for context such as "only allow this action if it's during business hours" or "only allow read access unless a higher privilege is specifically needed." It's up to the application or API to implement any such logic, which many do not. OAuth also has limited support for **incremental authorization** (asking for more permissions later) – some providers support it, but it's not a universal feature. Therefore, applications often request the maximum scopes up front "just in case," leading to over-broad initial grants. There is also typically **no partial revocation** of scopes: if a user or admin wants to revoke one permission, they often must revoke the entire token, cutting off all access until a new token with a reduced scope is issued. This was identified as a major shortcoming of JWT-based access tokens – privileges remain until the token expires, even if an admin changes the user's rights in the meantime [8]. In practice, OAuth tokens act like coarse session credentials that are difficult to rein in once unleashed.

Token Theft and Exposure Risks

OAuth's security also hinges on the secrecy of the access token. **Bearer tokens** (which OAuth uses by default) are like passwords: anyone who possesses the token can use it to access the associated resources, until it expires or is revoked. This makes OAuth tokens a high-value target for attackers. If tokens are stolen – through malware on a user's machine, a leak from an app's database, or a vulnerability in a third-party integrator – an attacker can impersonate the legitimate application or user. Unlike passwords, users are not prompted when tokens are used, so theft can go unnoticed until damage is done. A striking real-world example occurred in **April 2022**, when GitHub discovered that attackers had **stolen OAuth tokens issued to two third-party apps (Heroku and Travis CI)** and abused them to access data from dozens of organizations' private GitHub repositories [9]. Because those tokens carried broad `repo` permissions, the attackers were able to download source code from private repos en masse. GitHub's analysis noted the threat actors likely mined these repositories for secrets (like API keys) to pivot into other systems [10] – a classic "breach upon breach" scenario enabled by a single OAuth token compromise.

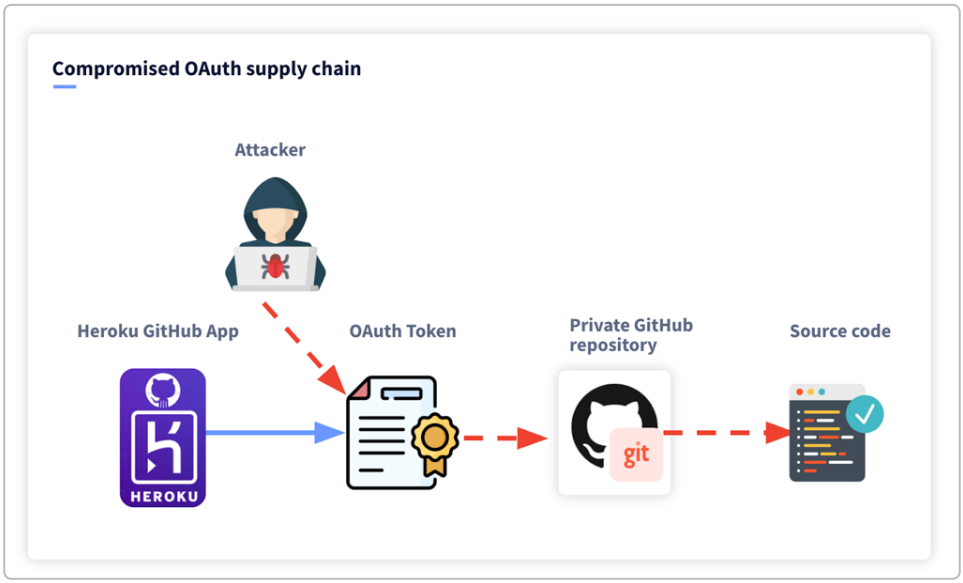


Illustration of a compromised OAuth supply chain: an attacker steals the OAuth token from a third-party integrator (e.g., a Heroku GitHub app) and uses it to retrieve private repository data from GitHub [11]. In the 2022 GitHub incident, the stolen tokens effectively granted the attackers the same privileges the legitimate apps had. GitHub confirmed that these **compromised tokens were used to clone private repositories from dozens of victim organizations** [12]. This incident highlights how OAuth tokens, if

not adequately protected, can become a single point of failure: one token in the wrong hands led to a supply-chain breach across many companies. The risk of token theft is exacerbated by inconsistent security practices in storing and handling tokens. Developers may inadvertently log tokens, or store them in plain text in databases, or not deploy proper TLS/certificate pinning – any of which could expose tokens to interception. Additionally, long-lived refresh tokens (used to get new access tokens) present another attractive target. If an attacker gains a refresh token, they can continue to mint new access tokens even after the old ones expire, extending the window of compromise.

Beyond outright theft, OAuth tokens can be abused in other ways. Phishing attacks may trick users into authorizing a malicious app, effectively *giving* the attacker a token with the user's consent. This technique bypasses the need to steal credentials – the user willingly clicks "Allow" on a fraudulent OAuth consent screen. A notorious example is the **Google Docs OAuth worm** of 2017. In that campaign, attackers created a fake app named "Google Docs" and distributed consent requests that appeared to come from a trusted contact [13] [14]. Users who approved the request unknowingly granted the fake app permission to "Read, send, delete, and manage" their Gmail, as well as manage their contacts. The app then used those privileges to self-propagate – reading the victim's contacts and emailing itself to them – and to potentially access the victim's email data. Crucially, this OAuth-based attack **did not require the user's Google password or bypass any two-factor authentication**, since the user voluntarily authorized the access token [15]. It demonstrates that token misuse can be as dangerous as token theft. In both cases, OAuth's design means the authorization server will accept the token as proof of delegated rights, with no native way to distinguish malicious use from legitimate use of a stolen or fraudulently obtained token.

Implementation Inconsistencies and Platform Variations

Another security concern with OAuth is the **inconsistency of implementations across different providers and applications**. OAuth 2.0 is a framework with flexibility, which has led to variations in how it's implemented and extended. Not all OAuth providers support the same set of features or enforce the same security measures. For example, some providers may not strictly validate redirect URIs or may allow wildcard callbacks, leading to open redirect vulnerabilities. Others might skip implementing PKCE (Proof Key for Code Exchange) in certain flows, making public clients vulnerable to authorization code interception. Additionally, the definition and granularity of scopes vary widely. Google's OAuth scopes are very granular for some services and very broad for others (e.g., a single scope like `https://www.googleapis.com/auth/cloud-platform` grants access to *all* Google Cloud services under a user's account). In contrast, some services have only a few coarse scopes because their API doesn't support finer segmentation. This inconsistency makes it hard to adopt a uniform least-privilege strategy across all integrations.

In some cases, OAuth's flexibility has led to logic flaws when multiple identity providers or apps are involved. A recent security report (2025) detailed an OAuth flaw where an access token issued for one application could be **reused to access another application** from the same provider under certain misconfigurations [16]. Essentially, if two apps trust the same identity provider (say, Google) and the identity provider does not properly bind tokens to the intended audience, an attacker could take a token granted to App B and present it to App A to gain unauthorized access. This kind of implementation mistake – not mandating a token audience or client check – is not part of the core OAuth spec but is a recommended best practice. When implementers diverge from best practices, vulnerabilities like OAuth token "mix-up" or confused deputy problems can arise. The result is that the security of OAuth in practice is only as strong as the weakest link among the various implementations. An organization might use multiple OAuth-backed integrations (Google, GitHub, Slack, etc.), and each may have different default behaviors, timeout durations, or token formats. This patchwork can lead to unforeseen gaps.

For instance, consider the process of revoking access. There's no single standard for how a token can be programmatically revoked across providers – one API might let users or admins revoke tokens via an endpoint, while another requires manual action. If a user leaves an organization, ensuring all their third-party OAuth tokens are revoked is a challenge. A recent disclosure in late 2023 highlighted a scenario where **former employees retained access to corporate apps via OAuth tokens** even after their primary accounts were deactivated [17]. In that case, users had logged into services like Slack and Zoom using "Login with Google." When their Google Workspace account was suspended, the OAuth tokens and sessions in the third-party apps were still active because the connection was not fully severed. The **lack of a coordinated token revocation or visibility** meant admins could not easily even see these lingering tokens [17]. This example underscores how inconsistent handling of OAuth across systems (Google vs. Slack in this case) can create security blind spots. In summary, the OAuth ecosystem's non-uniform implementations and optional features can introduce security issues that are not immediately obvious, especially when an application – or an AI agent – straddles multiple services.

Implications for MCP and AI System Integrations

When we apply these OAuth limitations to the **Model Context Protocol** environment, the stakes become higher. By design, MCP allows an AI model to operate across **multiple systems and data sources**, maintaining context and performing actions on behalf of a user. If OAuth is the mechanism granting access to each system, then all the earlier concerns are multiplied by the number of integrated systems. Below we analyze the key implications:

- **Privilege Aggregation and Escalation:** Under MCP, a single AI model might hold OAuth tokens for a code repository, a cloud storage service, an email account, and more – all at once. Each token by itself may already be over-privileged (coarse scopes), and in aggregate the model effectively has the union of all these privileges [18]. This creates a *centralized vulnerability*. If the model is compromised (say, via a prompt injection that causes it to leak or misuse a token), an attacker could gain a **super-token** access to many systems at once. Even without an external attacker, the model itself could accidentally use a token in an unintended way. Since OAuth doesn't enforce contextual restrictions, the AI might apply a permission in the wrong context – for example, using an admin-level cloud API token when it only needed read access for a particular task. The difficulty of enforcing least privilege in OAuth means an MCP-integrated model is likely running with more privilege than necessary on each system [19] [20], and any bug or manipulation can escalate into a full breach of those systems.

- **Cross-System Exploitation:** In a traditional single-app scenario, if an OAuth token is compromised, only the connected service is affected. In MCP, however, an AI bridging multiple services could be a conduit for crossing over. An exploit in one system could be leveraged to attack another through the model. For example, if an attacker compromises the model's token for a chat platform, they might inject a malicious command that causes the model to use its database token to exfiltrate sensitive records. Since the model has credentials for disparate systems, a foothold in one can be used to move laterally into others – a form of **cross-system attack** unique to multi-integrated AI agents [21]. This is especially concerning if the model confuses context; the AI might not have human-level judgment about when not to mix data between systems. OAuth provides no built-in defense against such misuse: the token for Service A will happily be honored by Service A even if the AI was tricked by something from Service B. Thus an **MCP context can amplify the impact of any single OAuth token compromise** or mis-issuance, turning it into a multi-system breach.

- **Auditing and Accountability Challenges:** OAuth tokens don't inherently carry user intent information on each request – they simply grant access. In an MCP scenario, an AI model might make API calls to various services using tokens on the user's behalf, but if something goes wrong, it can be hard to trace *why* it did so. Was a certain action (e.g., deleting a file or leaking a document) the result of a legitimate instruction, a prompt injection, or an outright compromise? Traditional logs at each service will just show that the authorized token performed the action. Without fine-grained scopes or contextual authorization, we lack visibility into what the model was *supposed* to do versus what it actually did. This makes security auditing difficult [22]. The coarse nature of OAuth scopes also means logs won't show which specific subset of data the model was meant to touch. From a governance perspective, this is troubling: organizations might find it challenging to demonstrate compliance with data regulations (like proving that an AI only accessed permitted records) when OAuth tokens grant broad access and the AI's usage is a black box. In essence, OAuth wasn't designed for multi-hop delegated actions, which is what MCP orchestrates, so the usual assurances (scopes, consent) are too vague to capture the nuance of an AI's behavior across systems.

- **Compliance and Control Gaps:** Many industries have strict access control requirements, such as **separation of duties** and **need-to-know access**. OAuth's model of broad tokens and lack of partial consent can clash with these requirements. In MCP, an AI could unintentionally violate compliance by drawing data from one context into another. For example, an AI with access to both a healthcare database and a public code repository might inadvertently include sensitive patient data in a code comment pushed to GitHub – all because it had the OAuth credentials to do both and wasn't restrained in context. Regulators would rightfully be alarmed at such a scenario. The **lack of granular revocation** is another issue: if an anomaly is detected, simply cutting off the AI's access selectively is hard – one might have to revoke all tokens and thereby halt all functionality, or otherwise leave potential backdoors open. Organizations using MCP in regulated environments (finance, health, government) could face an **all-or-nothing choice** on trust: either trust the AI integration completely (with broad OAuth tokens), or disallow it entirely – a dilemma rooted in OAuth's coarse control model [23]. Until OAuth mechanisms improve, this could slow MCP adoption in sensitive fields, as noted by security perceptions that OAuth-based integrations might be too risky [24].

In summary, the very design decisions that made OAuth simple and flexible – static scopes, bearer tokens, broad consent – become critical weak points when an AI agent is orchestrating activity across many platforms. The Model Context Protocol promises to empower AI with seamless access to tools and data, but if it leans on OAuth without additional safeguards, it inherits all of OAuth's weaknesses. An MCP-integrated AI essentially concentrates the combined authority granted by multiple OAuth tokens, and any failure of that system (through attack or error) could have far-reaching consequences. This means that security practitioners must be extremely vigilant in how OAuth is used within MCP: monitoring token usage, employing anomaly detection, and putting external guardrails around the AI's actions will be crucial to prevent abuse.

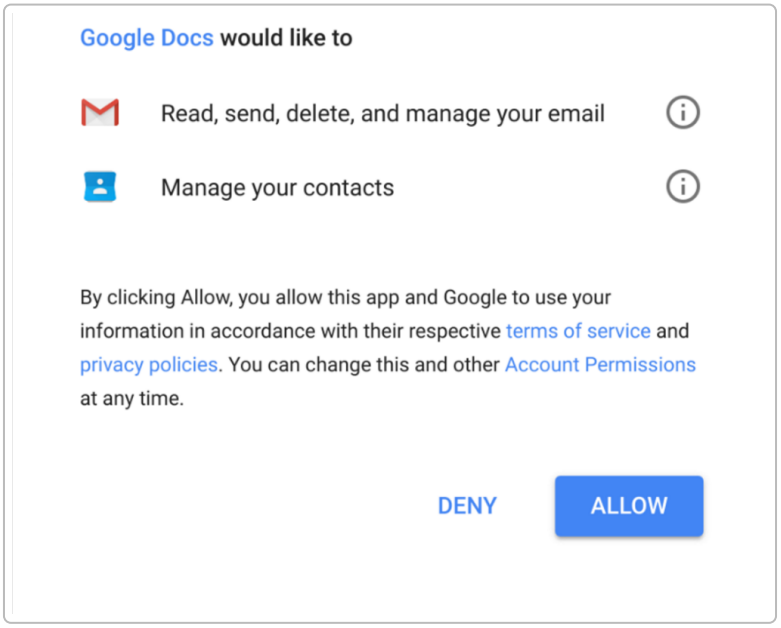
Real-World Case Examples of OAuth Vulnerabilities

To ground the discussion, we now examine a few real incidents and scenarios that exemplify OAuth's security issues, especially as they relate to multi-system integrations akin to MCP:

- **GitHub OAuth Token Breach (2022):** As introduced earlier, GitHub revealed that attackers stole OAuth tokens issued to popular DevOps integrations (Heroku and Travis CI) and used them to access GitHub's API on behalf of numerous victims. The stolen tokens granted what was

effectively *full repository access* to the victims' code (due to the broad scopes those apps required). GitHub's investigation found that dozens of private repositories were downloaded without authorization [12]. This case shows the impact of **token theft**: one leaked credential from a third-party service led to a supply chain compromise of many organizations. It also highlights the danger of **coarse scopes** – the Heroku and Travis apps requested extensive repo permissions, so the tokens the attackers obtained were extremely powerful. GitHub and the affected providers had to revoke all tokens and notify users to reauthorize, a disruptive process. If an AI system were using MCP to interface with GitHub under the hood, such a token theft could mean an attacker suddenly has the AI's privileges. For instance, an attacker could silently use a stolen token to query or modify data, and the AI (or the user) might not realize that its token is being misused elsewhere. This incident underscores why keeping OAuth tokens secure is paramount, and why **over-privileged tokens** (with "all repo" access) are ticking time bombs.

- **Google Docs OAuth Worm (2017):** This incident stands out as an OAuth-driven attack that spread rapidly. The malicious application cleverly named "Google Docs" leveraged Google's OAuth consent process to get users to approve access to their Gmail and contacts [25]. Thousands of users clicked Allow, perhaps thinking it was a legitimate Google request, given the app's name and the context of receiving an email from a known contact. Once authorized, the app had the ability to read and send emails and harvest contacts – essentially full email account control. The worm then emailed itself to all of the victim's contacts, using Gmail's API via the OAuth token, and hid the sent messages. Not only did this attack propagate quickly in a **worm-like fashion** [26], it also demonstrated a crucial point: OAuth can be an **attack vector** that bypasses traditional security. Even strong passwords and 2FA were ineffective here, because the user's **consent** was the only barrier, and it was overcome by deception [15]. For MCP, this example is a warning that user consent for AI actions must be treated carefully. If a malicious request made it into an AI's instruction (say via a poisoned data source) and the AI had an OAuth token, the AI might perform a dangerous action thinking it's authorized. The Google Docs worm also emphasizes the need for better app verification – Google later improved their processes to prevent apps from impersonating Google's own product names and to alert users to unverified apps. In the context of MCP, ensuring that the AI only uses OAuth tokens with verified, trusted services is vital; otherwise, the AI could be tricked into interacting with a malicious service.



A deceptive OAuth consent prompt from the 2017 Google Docs phishing attack. The fake app "Google Docs" requested broad Gmail and contacts permissions, offering only a single Allow/Deny choice to the user (no granular consent), which many unwittingly accepted. This real-world phishing incident demonstrates how OAuth's **lack of selective authorization** and user interface limitations can be exploited. The permissions requested (as shown above) were extremely broad – "Read, send, delete, and manage your email" basically grants full control over the Gmail account. Yet the user had no option to grant lesser access (they could not, for example, say "just read emails, not delete or send"). The **coarse consent model** made it an all-or-nothing decision, and many chose "Allow," enabling the worm to abuse their accounts [25]. For cybersecurity professionals, this case is a reminder that even absent a technical vulnerability, the OAuth flow can be manipulated through social engineering and UI design. In an AI scenario, one might imagine an attacker crafting input to an AI assistant that causes it to visit a malicious OAuth link – if the AI is operating autonomously with certain credentials, it could potentially self-authorize a bad request. While that is speculative, it's analogous to how users were tricked; an AI agent with the ability to initiate OAuth flows would need safeguards to avoid "Allowing" actions that compromise security.

- **OAuth Implementation Flaw in Google Cloud Integrations:** There have also been reports of more subtle OAuth issues that could affect enterprise cloud environments. In late 2023, researchers highlighted a Google OAuth oversight that allowed employees (or ex-employees) to retain access to third-party SaaS accounts even after their Google Workspace access was suspended [17]. By using a personal Google account masquerading as a corporate account (via an email alias trick), an individual could sign into services like Slack using OAuth, and that session would not be tied to the actual corporate user identity. When the person left the company and their official Google account was deactivated, the Slack (or other SaaS) OAuth session persisted because it was linked to the attacker-controlled Google account which remained active [17]. This is less of a single dramatic incident and more of a structural vulnerability, but it showcases an **implementation inconsistency** with serious security implications. The identity provider (Google) and the service (Slack/Zoom) had a trust model that could be gamed, and there was no visibility to administrators. In the context of MCP, this translates to a need for vigilance about how tokens are issued and tied to identities. If an AI is granted a token, is it clear *whose* authority it carries, and is it automatically revoked when that user leaves or when the AI's authorization is no longer valid? Currently, OAuth leaves these questions to each implementation. A mismatch or oversight (as in the Google-Slack example) can create a **backdoor that stays open**. MCP integrators should ensure that there are processes to audit and expire tokens appropriately, or else an AI could continue to have access that no one realizes it has.

These examples underscore the multifaceted nature of OAuth security problems: blunt permission scopes enabling overreach, token theft enabling large-scale breaches, user consent UX enabling phishing, and implementation nuances enabling persistent access or confusion. Each of these is highly relevant to any system, like MCP, that will rely on OAuth to mediate access between an AI and external services. The incidents with GitHub and Google also show that **these are not just theoretical concerns** – attackers are actively targeting OAuth credentials and flows to compromise systems. Therefore, anyone deploying AI systems that integrate with other platforms must treat OAuth tokens and permissions as critical security assets.

Conclusion

OAuth has been a powerful enabler of connected applications, but its **current limitations present significant security challenges** for advanced integrations such as the Model Context Protocol. As we have discussed, OAuth's coarse-grained permission model and all-or-nothing consent provide a poor

level of least-privilege control. Users (or AI acting on users' behalf) are often forced to grant broader access than necessary, increasing the potential impact if something goes wrong. The inability to selectively authorize or easily revoke specific permissions means that once an OAuth token is issued, it becomes a wide-ranging key that is hard to constrain. This is especially dangerous in the MCP scenario, where an AI may hold multiple such keys simultaneously. We have seen how **token theft or misuse can result in major breaches** – for example, a single stolen token led to dozens of companies' source code being downloaded [12], and a single malicious app's token led to a worm that spread across countless Gmail accounts [25]. In an AI context, a compromised token could allow an adversary not only to access data but potentially to manipulate the AI's behavior (by feeding it falsified data from a system it trusts, for instance).

The **implementation inconsistencies in OAuth** add another layer of risk. MCP-based systems will interact with multiple OAuth providers, each with their own quirks and potential vulnerabilities. As the examples illustrated, any weak link – whether it's a lax redirect URI check, an overly permissive scope, or a token that isn't revoked when it should be – can become the vector for an exploit. Unfortunately, when an AI is in the middle, that exploit can propagate through the very connections that MCP is meant to facilitate, undermining the integrity of the AI's operations across *all* connected systems.

In conclusion, the **security concerns of OAuth have outsize implications for AI integrations using protocols like MCP**. Organizations experimenting with connecting AI models to their sensitive systems via OAuth need to be acutely aware of these risks. The severity of potential impacts ranges from loss of sensitive data, to unauthorized transactions, to complete compromise of multiple linked accounts. While this paper has focused on analyzing the problems rather than solutions, the clear takeaway is that **additional safeguards are necessary** when using OAuth in such contexts. This might include tighter scope management, rigorous monitoring of AI actions, frequent token audits, user education on consent, and possibly augmenting OAuth with custom restrictions or verifying steps for AI usage. Absent such measures, the convenience of OAuth could turn into a nightmare scenario in which an AI meant to increase productivity instead becomes an unwitting conduit for security failures. The OAuth protocol's limitations, if unmitigated, could significantly weaken the security posture of MCP-enabled AI systems – a risk that cybersecurity professionals and AI developers must not ignore.

---

[1] Introducing the Model Context Protocol \ Anthropic
https://www.anthropic.com/news/model-context-protocol

[2] GitHub oAuth apps: What's the most granular scope to get access to Pull Requests? - Stack Overflow
https://stackoverflow.com/questions/74314624/github-oauth-apps-whats-the-most-granular-scope-to-get-access-to-pull-requests

[3] git - Minimal set of scopes to push to github using an access token - Stack Overflow
https://stackoverflow.com/questions/63906613/minimal-set-of-scopes-to-push-to-github-using-an-access-token

[4] Google Introduces Consent Unbundling for OAuth in Ads APIs Starting May 2025
https://web.swipeinsight.app/posts/consent-unbundling-for-oauth-user-authentication-in-google-cloud-apis-14891

[5] [6] [7] [8] Using OAuth 2.0 scopes vs. permissions for app authorization
https://www.aserto.com/blog/scopes-vs-permissions-authorization

[9] [10] [12] Security alert: Attack campaign involving stolen OAuth user tokens issued to two third-party integrators - The GitHub Blog
https://github.blog/news-insights/company-news/security-alert-stolen-oauth-user-tokens/

[11] How Hackers Used Stolen GitHub Tokens to Access Private Source Code
https://blog.gitguardian.com/how-hackers-used-stolen-github-oauth-tokens/

13 14 15 25 26 Gmail OAuth Phishing Goes Viral | Duo Security
https://duo.com/blog/gmail-oauth-phishing-goes-viral

16 One Token, Two Apps: The OAuth Flaw That Can Compromise Your Accounts — A Silent Security Disaster | by Rahul Gairola | Medium
https://medium.com/@rahulgairola/one-token-two-apps-the-oauth-flaw-that-can-compromise-your-accounts-a-silent-security-disaster-31cff04dcceb

17 2024 Google OAuth Vulnerability Technical Guide | Nudge Security
https://www.nudgesecurity.com/post/google-oauth-vulnerability

18 19 20 21 22 23 24 mcp_oauth_security_brief.md
file://file-ScmbB61bFngWFM6VwvLK3F