



2026-10-01

WHITEPAPER

Demystifying Passkeys: Under Attack

PRESENTED BY: [Matteo Giordano](#)

Anvil Secure

2125 Western Ave Suite 208

Seattle, WA 98121

United States of America

+1 206.753.7649

info@anvilsecure.com

Table of Contents

Introduction	5
Two Planes: Research and Engagement	7
Actor 1: The Authenticator and the Protocols	8
Formal Security Analyses	8
Algorithm Downgrade	8
Hardware vs Software	9
The Silicon	10
CTRAPS: Eleven Attacks Against the Management API	11
What the Relying Party Inherits	12
Actor 2: The Hybrid Transport	13
Limits of Relay Detection	15
HiPass: Experimental Results	15
PoisonSeed: Attempted Cross-Device Relay	16
The FIDO: / Intent Bug (CVE-2024-9956)	16
Linked Devices	16
What a Relying Party Can Actually Do	17
Actor 3: The Client	18
Forging the Whole Ceremony, AAGUID Included	18
Signed Assertion Hijacking	19
The Password Manager Is a Second WebAuthn Implementation	20
What Actually Defends This	20
Actor 4: The Relying Party	21
Decoding and Re-encoding WebAuthn Traffic	22
Origin and rpIdHash Validation	25
Related Origins and Dangling Domains	26
The Flags Byte	27
COSE Algorithm Downgrade and Confusion	28
Challenge Freshness and Session Binding	28
Registering a Key on an Account You Don't Own	28
Credential IDs and the Overwrite Class	29
Attestation Policy and Library Verification Failures	29
Signature Counters and RP Policy	30
Checklist and Creativity	31
Actor 5: The Sync Fabric	32
Trust Dependencies in Synced Passkeys	33
Three Routes Into the Sync Account	33
Vault Cryptography and Backup-State Validation	34

Compromise Detection and Emerging Defenses	35
What Falls Inside a Normal Scope	35
Actor 6: The User and Recovery	36
The Mixed-Mode Problem	36
The Downgrade Path	37
Three Ways Around the Ceremony	38
User Verification and Biometric Enrollment	39
Interpersonal Threat Model	39
What the Person Holding the Passkey Is Left With	40
Test-Case Reference	41
The Tooling	52
Conclusion	54
References	55
About the Author	59
About Anvil Secure	60

About This Series

A three-part series on passkeys for security engineers and offensive security specialists. You are reading the **third part**.

1. **Under the Hood: The Protocol**. How passkey ceremonies work at the byte level.
2. **Under the Hood: The Architecture**. Distinctive features, taxonomies, and deployment edge cases.
3. **Under Attack**, this paper. Testing and attacking the six components touched by passkeys.

Introduction

Passkeys replace reusable passwords with public-key credentials and resist conventional credential phishing. Their security also depends on the systems surrounding the registration and authentication ceremonies.

This paper examines that attack surface as of 2026, covering the six actors involved in passkey deployments and distinguishing research findings from deployment-level testing.

Passkeys, meaning FIDO2 discoverable credentials whether they stay on one device or sync through a provider, do effectively eliminate traditional credential stuffing and simplistic Adversary-in-the-Middle (AiTM) phishing. There are at least four categories of passkey, as we'll see later, each of them moving the trust boundary somewhere else, so the threat model shifts with the category and they don't all inherit the same attacks. Wherever only one category is in play we'll name it, whereas plain *passkeys* means all of them. Pentesting passkeys means auditing every actor the ceremony depends on, because a sign-in is only ever as strong as the weakest of them, and that list runs a long way past the cryptography.

If you haven't already, read [the first part of this series](#), where we dissect the protocol stack, the registration and authentication ceremonies, and the raw `authData` byte structures that fly across the wire. This paper assumes you're comfortable with the `flags` byte, `clientDataJSON` origin binding, and the difference between device-bound and synced passkeys.

To put a number on the scale we're working with: as of the FIDO Alliance's *State of Passkeys 2026* report, there are roughly 5 billion passkeys in active use, 90% consumer awareness, and 75% of consumers have enabled passkeys on some accounts. On the workforce side, 68% of organizations are deploying or piloting passkeys. But only about 30% use passkeys as the *primary* workforce sign-in, and roughly 57% still rely mainly on passwords or other phishable methods ([FIDO Alliance, The State of Passkeys 2026, May 2026](#)).

That gap between "passkeys are deployed" and "passkeys are the only way in" is blurry, and in many cases it's the way attackers put a foot in the door.

FIDO's own CTO makes half of that argument for us in [Passkeys Are Not Broken. The Conversation About Them Often Is](#) (Nishant Kaushik, September 2025), which defends the ceremonies and points at the environment around them, the components this paper walks through, as the fallible part and the part attackers go for.

We'll follow a handy mental framework to organize what's in the wild:

1. **The authenticator and the protocols.** WebAuthn and CTAP as specified, formal verification, silicon, and the CTAP management surface.
2. **The hybrid transport.** Cross-device authentication unexpected flaws.
3. **The client.** Browser, extension, password manager. Assume-breach territory.
4. **The relying party.** The heaviest stop, where a time-boxed engagement spends nearly all its hours.
5. **The sync fabric.** Credential synchronization, sharing, export, and provider security.
6. **The user and recovery.** Fallbacks, enrollment, revocations and inter-personal threat models.

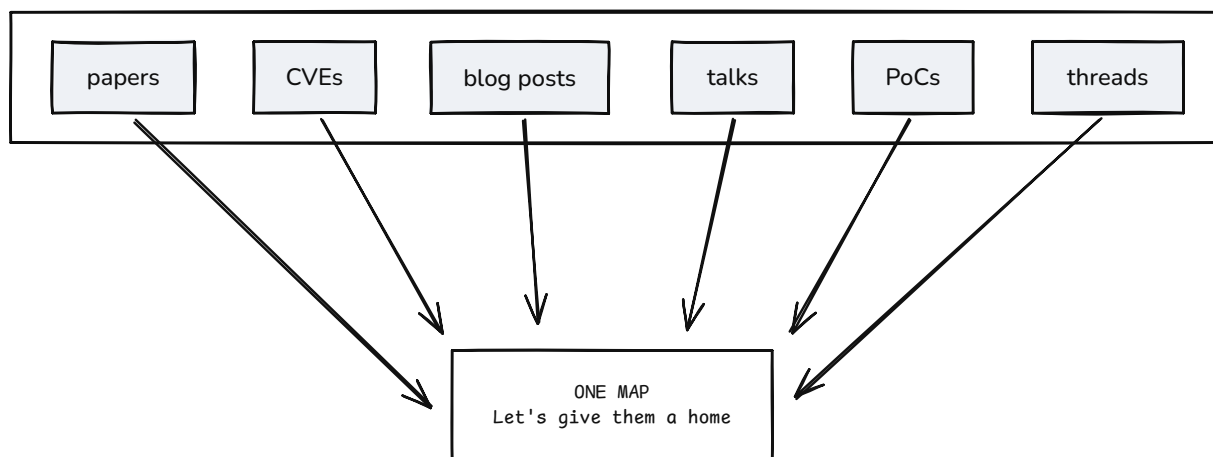


Figure 1: Scattered Research, One Map

Two Planes: Research and Engagement

There are two jobs you can do against this surface, and they overlap less than one might expect:

- **Research** picks one actor and goes as deep as it takes, and it's finished when the class is proven.
- An **engagement** is time-boxed against one deployment, and it's finished when the scope is covered and somebody has a report they can act on.

	Research plane	Engagement plane
Shape of the work	Pick one actor, go as deep as it takes	Time-boxed: a bounty, a pentest, a red team gig
What you produce	CVEs, 0days, PoCs, papers	Findings against one deployment, with a report deadline
What "done" means	The class is proven	The scope is covered and the client knows what they shipped
Where it lands here	Actors 1, 2, and much of 3	Actor 4, plus 5 and 6 on a red team

Most published passkey work sits on the research plane, which is why the public record is heavy on beautiful hardware attacks and thin on "here's what to test when you have nine days and a login page." Hence, this paper tries to be more balanced across all actors.

Actor 1: The Authenticator and the Protocols

Who	What	How
Spec authors at the W3C and the FIDO Alliance, silicon vendors, and the academic cryptographers who audit both.	WebAuthn and CTAP as written, the CTAP management API, and the chip the private key lives on.	Proofs against the specifications, probes and glitches against the chips. Research plane , and the output is theorems and the occasional silicon CVE.

Two protocols carry the whole thing, WebAuthn between browser and server and CTAP between the client and a roaming authenticator, and both have been picked over by cryptographers for years with the core ceremony still standing under the assumptions those analyses fix in advance.

The people working down here run symbolic model checkers and hand-built computational proofs against the specifications, and electromagnetic probes and voltage glitches against the chips, publishing theorems and the occasional CVE against a piece of silicon; almost none of it can be reproduced on a deadline, so what follows is a map of what has already been settled to give an idea of the state of the art, rather than a set of tests to run.

Formal Security Analyses

[Barbosa, Boldyreva, Chen and Warinschi](#) gave the first provable-security analysis of FIDO2 at CRYPTO 2021, covering CTAP 2.0 and WebAuthn 1; [Bindel, Cremers and Zhao](#) extended it to CTAP 2.1 and WebAuthn 2 at IEEE S&P 2023, pulling token binding, none attestation, user verification and user presence into the model; [Barbosa and co-authors](#) revisited privacy and security at PoPETs 2025; and attestation got its own treatment from [Bindel, Gama, Guasch and Ronen](#) at ASIACRYPT 2023. All four are computational proofs. The symbolic side reads less uniformly positive, since [Guan et al.](#) modeled the CTAP2-plus-WebAuthn composition in ProVerif at ESORICS 2022 and found that some strong authentication properties fail to hold, along with weaknesses in the Client PIN mechanism, so "the protocol is proven" is a sentence that needs a qualifier attached to it.

The S&P 2023 paper also produced the result that travels furthest, the first formal guarantees for the none attestation mode that consumer deployments use by default, which turns out sound for authentication while saying nothing about which authenticator signed, so an unverified AAGUID cannot substantiate a model restriction.

Each of those proofs assumes an implementation that follows the spec, and each one covers the ceremony and nothing around it, so a deployment can follow the spec exactly and still lose to behavior no model ever looked at. Most of the rest of this paper is a catalog of implementations that don't follow the spec, or (like CTRAPS, which we'll see in a bit) of attacks on something adjacent to the core ceremonies.

Algorithm Downgrade

At registration, the relying party advertises acceptable algorithms in `pubKeyCredParams`. Rewriting that list before it reaches the authenticator requires a compromised client or registration channel, for example a malicious browser or a TLS-intercepting proxy trusted by the client. With that access, an attacker can leave only a weaker algorithm that the RP advertised and the authenticator supports. The resulting credential remains

bound to that algorithm after the registration-time compromise ends. [Bindel, Cremers and Zhao](#) documented it in the same paper that proves WebAuthn 2 and CTAP 2.1 secure, proposing a fix they call WebAuthn 2+, in which the authenticator hashes the list it actually received so the server can compare, which detects a downgrade without preventing one and which neither W3C nor the FIDO Alliance has adopted, leaving it a research proposal rather than a control anyone can ask a vendor about.

The list this attack rewrites is about to get much more interesting because of the post-quantum clock, which is what makes a detection-only proposal worth keeping in mind.

FIDO2 is unusually well placed for that migration, since the credential proof a relying party verifies is a signature over a fresh challenge and nothing else, so there's no stored ciphertext for a "harvest now, decrypt later" adversary to come back to and nothing to re-encrypt. The stack isn't quite free of Shor-breakable key establishment, because CTAP's PIN/UV protocol agrees its shared secret over ECDH, though that exchange takes local USB or NFC access to record and protects a PIN rather than anything with long-term value, and the same Bindel, Cremers and Zhao paper shows the step swaps out for a post-quantum KEM with minimal protocol changes.

What has to change on the WebAuthn side is the signature algorithm, and NIST finalized the replacements in August 2024, [FIPS 204](#) (ML-DSA) and FIPS 205 (SLH-DSA), with FIPS 206 (FN-DSA) still in draft as of mid-2026.

Nobody flips the whole ecosystem to post-quantum algorithms on one day, so for years `pubKeyCredParams` will offer a hybrid classical-plus-post-quantum entry *and* a classical entry underneath it, because the server can't know whether the authenticator in front of it understands the new one. That's a downgrade target by construction: strip the hybrid entry and the authenticator picks the classical fallback, having no way to tell it was ever offered anything better. Security keys stay in service five to ten years, so the population of authenticators that only speak classical will keep that fallback alive for the whole window, and the detection mechanism that would catch the strip is the one nobody adopted.

The separate [RP-side enforcement test](#) submits a registration response using an algorithm absent from the RP's original `pubKeyCredParams` and checks that the server rejects it. This test can use a tester-controlled client and account without compromising another user's registration channel.

Hardware vs Software

[The Architecture](#) split authenticators into platform, roaming and hybrid; cross that with first-party against third-party (iOS Keychain and Windows Hello, Microsoft Authenticator, an Apple or Google phone acting as roaming, a YubiKey or Bitwarden) and then cut once more into hardware and software.

A hardware authenticator generates the private key inside a secure element and exposes no interface for getting it back out, so the key is created in one place and stays there for the life of the credential; when that authenticator reports user presence and user verification, the bits are backed by a physical button somebody pressed and by a PIN or a fingerprint the element checked for itself.

A software authenticator keeps the key in ordinary storage, protected by whatever the operating system offers that day, and it sets those same two bits by assigning them, because nothing in the ceremony obliges a software implementation to have asked a human anything at all. The byte that reaches the server is identical in both cases, and nothing in it says which kind of authenticator produced it.

passkeys.dev keeps a [running known-issues list](#) of providers that "have not implemented User Verification in a spec-compliant manner," and the names on it aren't hobby projects (i.e., pull it live rather than trusting any snapshot including this one). Underneath sits the quieter problem, that a server can check UV=1 correctly every single time and still be told a story, because nothing in the ceremony distinguishes a real biometric prompt from a boolean somebody set to true. The spec's only lever here is restricting registration to approved authenticator models, rare enough in the wild that the relying-party section can put measured numbers on it.

The Silicon

Side channels are the famous lane, since ECDSA leaks the private key whenever the per-signature nonce repeats or leaks a few bits: NinjaLab's [A Side Journey to Titan](#) (2021) pulled keys out of Google Titan and other NXP A700x devices using electromagnetic analysis and roughly \$12,000 of gear, and [EUCLEAK](#) (2024, [ePrint 2024/1380](#)) went after Infineon's whole security-microcontroller family rather than one model, with outright cloning as the goal.

Fault injection is the other lane, glitching voltage or clock until an instruction skips a check, covered for FIDO2 by [Yadav and Seamons](#) (NDSS 2024) and familiar to us from [our own work on STM32 read-out protection](#).

Both lanes settle the same scoping argument, which comes up on nearly every engagement involving hardware keys: somebody asks whether the key itself can be broken, and the honest answer is yes, by a lab, given the key in hand for hours and a five-figure rig, which places this work primarily in specialist hardware assessments.

So don't spend the window there. The version of that question you can actually answer is about custody rather than physics, meaning where the key sits when it isn't plugged in, whether anyone ever set a PIN on it, and what happened to the spare that lives in a desk drawer.

And there's a much cheaper attack sitting underneath all of it. Point a reader at a key you found, send it an `authenticatorGetAssertion` command with no PIN and no fingerprint, and many keys will [answer](#), handing back the discoverable credentials they hold for the relying party you named, along with their credential IDs and user IDs. The command carries an RP ID, so what comes back is a yes or no on one candidate at a time rather than the key's whole address book, and you work through a list you brought with you, the domains you already care about first and the large consumer providers after them. Full enumeration through `authenticatorCredentialManagement` normally requires PIN/UV authorization to start, after which continuation requests use the established enumeration state. Deleting a credential also requires an authorized request.

The extension that governs this is `credProtect`, and the spec gives it three levels:

1. **`userVerificationOptional (0x01)`** answers anybody, and it's what an authenticator falls back to when nobody asks for anything stronger.
2. **`userVerificationOptionalWithCredentialIDList (0x02)`** answers only when the platform already supplies the credential ID, or when user verification happened, so a bare probe learns nothing.
3. **`userVerificationRequired (0x03)`** demands user verification either way.

Which level a credential lands at gets decided at registration by the client and the relying party rather than by the key, and [Chromium asks for 0x02](#) on any discoverable credential precisely so that holding a key doesn't reveal which sites it answers for, which makes the parking-lot path narrower than it first looks. It stays open wherever nobody asked and wherever the authenticator never implemented the extension, and Casagrande and Antonioli

still found production deployments sitting at the weak default with Microsoft and Apple among them, so this is worth probing rather than assuming in either direction.

CTRAPS: Eleven Attacks Against the Management API

Client

Authenticator

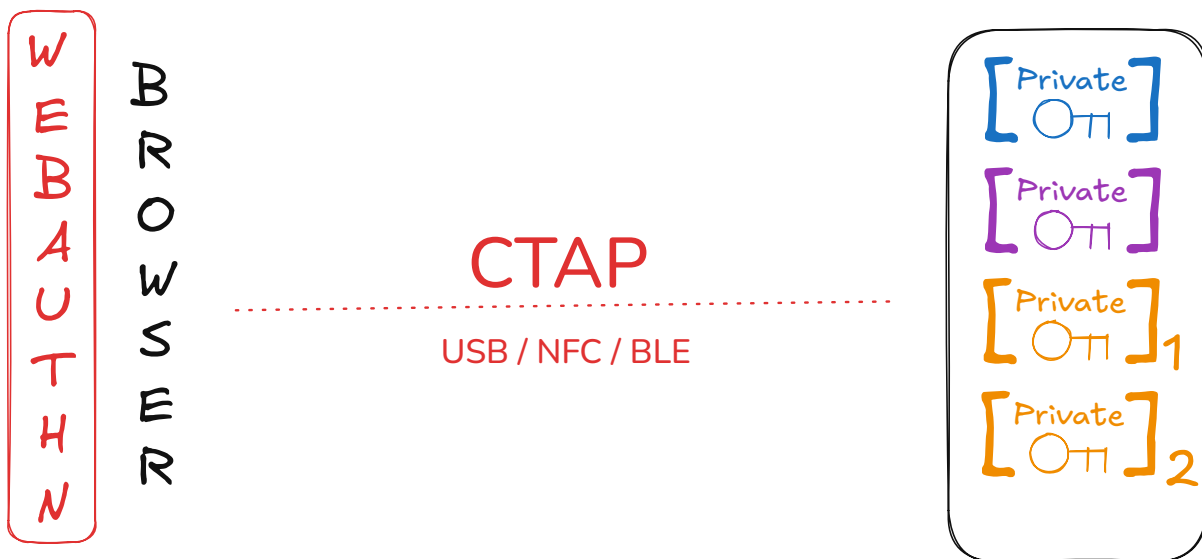


Figure 2: The CTAP Link

Beyond ceremonies, CTAP exposes a management API for creating, enumerating and deleting credentials and for factory-resetting the device, and [Marco Casagrande and Daniele Antonioli](#) (IEEE EuroS&P 2025) gave it its first security and privacy evaluation, drawing eleven attacks in two classes out of eight protocol-level vulnerabilities and proposing eight backward-compatible countermeasures alongside them.

Client Impersonation follows from CTAP never authenticating the client, so anything within NFC range can factory-reset a key with zero clicks (NFC range counts as user presence, so no button press ever happens), lock it by feeding wrong PINs, or pull CredId and UserId for tracking wherever the relying party left CredProtect at its weak default.

API Confusion needs a MitM between client and authenticator, whether an NFC skimmer, a hostile USB hub or a userspace shim over `node-hid`, and swaps one API call for another carrying the same or lower UP and UV requirements; AC1 is the one to remember, where the user believes they're authenticating while the attacker enumerates and deletes every discoverable credential invisibly, because the authenticator never reports which API actually ran.

All six authenticators on the bench proved vulnerable, as did all ten production relying parties tested, Adobe, Apple, GitHub, Microsoft, NVidia, Synology, DocuSign and Facebook among them, to credential deletion or

denial of service, with eight of them exposed to user tracking as well and the full rig published at github.com/Skiti/CTrAPs.

Two things bound how far this travels. The first is range, because the attacker has to be genuinely close to the key, 2 cm with a phone's NFC radio and 6.5 cm with a Proxmark3, which is nothing like the BLE distances in the next actor and rules out anything you could do from a parked car.

The second is that the disclosure record reads badly. FIDO called proximity attacks less scalable than remote ones and committed to nothing, Google closed the report on the claim that the attacks need a compromised client, which the authors dispute and are right on the facts, and Apple and Microsoft both said the weak CredProtect default had "no security implications for their products."

[CTAP 2.3](#) became a FIDO Alliance Proposed Standard on February 26, 2026 and includes the optional Long Touch for Reset feature. The `LongTouchForReset` field in `authenticatorGetInfo` reports whether the feature is supported and enabled. When enabled, reset requires a user-presence touch lasting at least five seconds; when supported but disabled, a single touch is sufficient.

The feature strengthens user-presence confirmation without adding user verification to the reset operation. It therefore doesn't implement the CTRAPS authors' proposed requirement to authorize reset with a valid PIN.

What the Relying Party Inherits

Two questions follow for deployment testing, architecture review and procurement.

Does the RP verify authenticator provenance before enforcing its model policy? The `attestationObject` carries an AAGUID at bytes 37 to 52 of `authData`, as described in [The Protocol](#). An allowlisted AAGUID is insufficient on its own: the RP must validate the attestation and establish that its trusted provenance supports the model claim. Accepting an authenticator is a finding only if it violates the deployment's documented policy.

Does it trust the UP and UV flags it's handed? Software authenticators set those bits without asking a human, several shipping providers do exactly that, and nothing downstream can tell the difference, so every risk decision made off those flags is one the server outsourced to the authenticator's word.

Actor 2: The Hybrid Transport

Who	What	How
The platform vendors who implement cross-device authentication, Google, Apple and Microsoft, on top of the FIDO spec.	Mostly the QR-plus-BLE handshake, where a single advertisement is the only thing binding the phone to the machine you're signing in on.	Wedge yourself between the two devices and play one of them, so the victim ends up signing you in. Research plane , and the relying party never sees a thing.

One link up from the silicon sits the transport that lets you use passkeys that live on a different device than the one you're signing on, which the spec calls **hybrid transport**, FIDO marketing calls **Cross-Device Authentication**, and Chromium source still calls **caBLE**.

[The Architecture](#) walked the mechanics of one of the most frequent use cases (i.e., QR+BLE): the client shows a QR code carrying a FIDO: / URI, the phone scans it and emits a single BLE advertisement so the two devices can prove they're near each other, and the real CTAP2 traffic then flows over an end-to-end encrypted tunnel through a cloud relay.

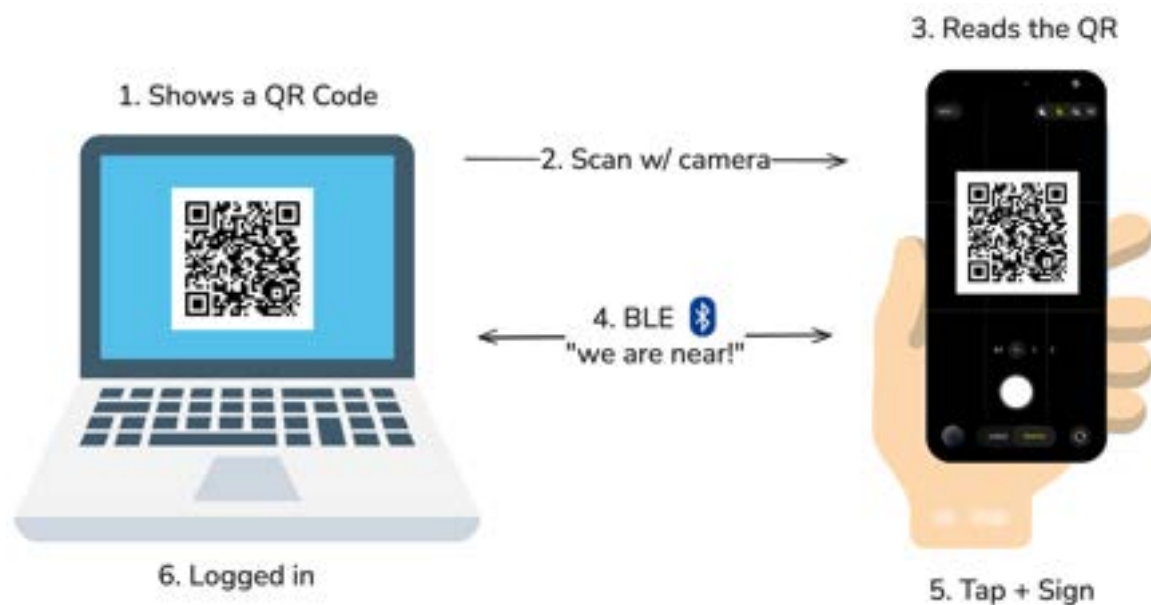


Figure 3: The Six-Step Cross-Device Flow

The entire security story rests on that advertisement, which proves only that your authenticator is in Bluetooth range of whatever client generated the QR code, never that the client is *yours*.

The attacker phishes the victim to a lookalike login page, then opens the *real* relying party in a browser of their own and starts a cross-device sign-in there, so the FIDO: / QR their browser draws is backed by a live ceremony at the legitimate origin. They forward that QR onto the phishing page, and when the victim scans it the

advertisement their phone emits reaches the attacker's machine, which is sitting inside Bluetooth range and is **the client that generated the code**, so the CTAP2 tunnel forms with the attacker's browser instead of the victim's own laptop.

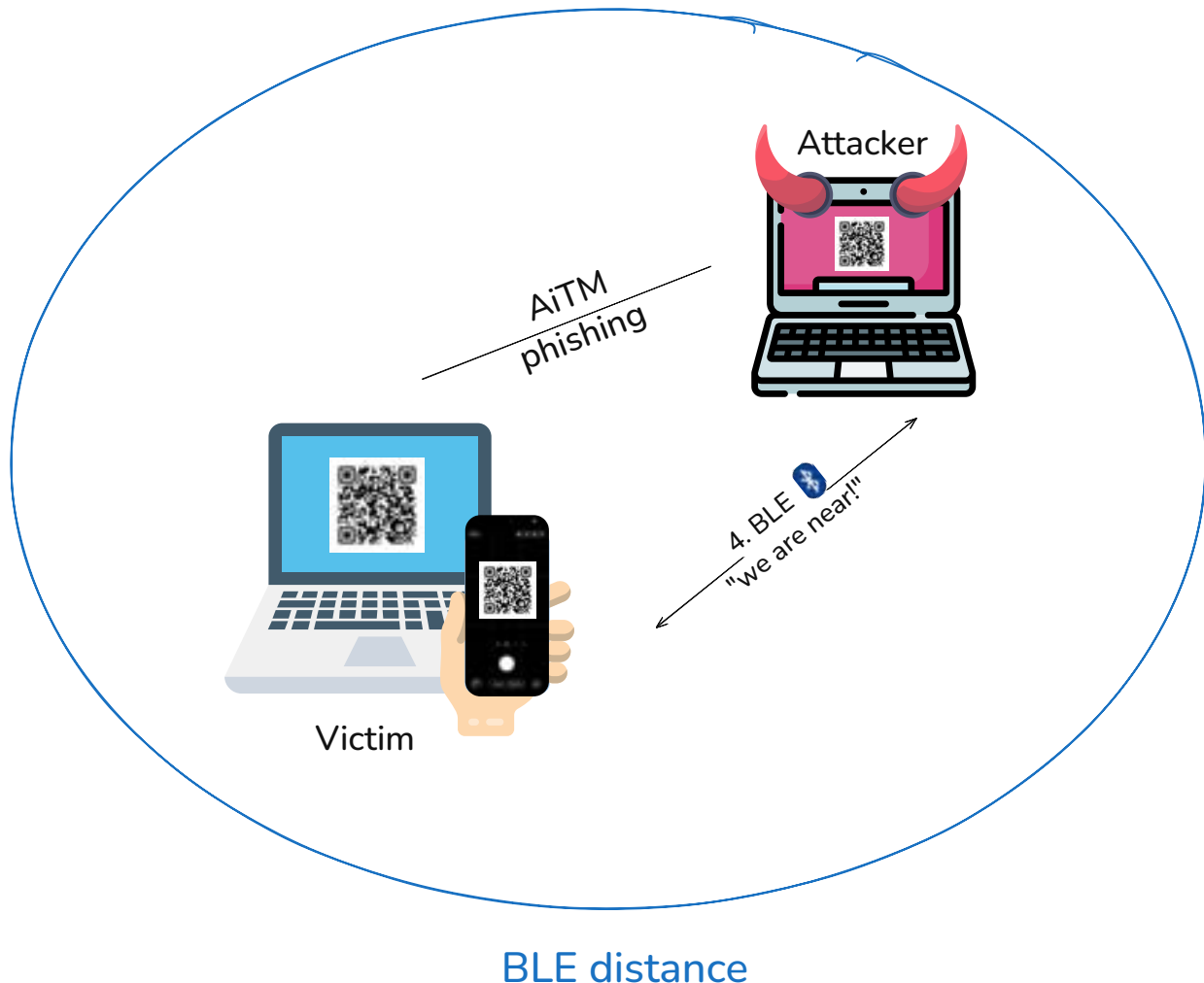


Figure 4: Attacker Inside BLE Range

Three groups who never coordinated have demonstrated that gap, inovex, Dennis Kniep and the peer-reviewed HiPass team, alongside a fourth and mechanistically distinct variant that browsers have since patched, and the shipping specifications still carry no fix for it. **Origin binding holds the whole time**, which is what makes the result interesting, since the QR only carries what's needed to open the tunnel while the RP ID the phone displays rides inside the CTAP request the attacker's browser pulled from the real site, so the domain check passes cleanly, the biometric prompt names exactly the right site, and the signed assertion lands in somebody else's session with nothing broken anywhere along the way and every component doing exactly what it was designed to do.

All of it is **proximity-bound, PoC-grade and hard to scale**, so nobody four days into a web-application

assessment is planting BLE beacons in a lobby, and where it actually pays off is in a physical red team, in a targeted social-engineering campaign, and in the threat-model conversations.

Limits of Relay Detection

The FIDO: / URI in the QR carries everything the authenticator needs to open that CTAP2 tunnel, which is exactly why forwarding it works, and [inovex](#) ran it that way with a device parked in BLE range of the victim.

Restricting transports in `allowCredentials` doesn't help either, because the spec [calls that field a hint](#): all it does is tell the victim's browser which transports to offer in its own dialog, so setting it to `internal` takes the scan-a-QR option off that dialog and nothing else. It isn't signed, the attacker's client reads the list and ignores it, and an engineer who tells you cross-device sign-in is off because `transports` is locked down has configured **a hint rather than a control**.

The UK NCSC carries the finding with government framing, warning that malware on a device in physical proximity can tunnel the Bluetooth component and "could modify the destination site, causing the user to connect to a different site than intended," while calling it proof-of-concept research with no documented in-the-wild use. It cites [Dennis Kniep's PoC](#), which forwards the victim phone's BLE advertisement from beacons within roughly 150m and **defeats the proximity check outright**, where HiPass below assumes consumer Class 2 range nearer 10m.

Nobody has found a way to make the transport answer the client question, and through [WebAuthn Level 3](#) and [CTAP 2.3](#) the specification never asks it.

HiPass: Experimental Results

[inovex](#) and [Kniep](#) showed the attack works; a team from Soongsil University and Kongju National University measured it and put the results through peer review in [HiPass: Hijacking CTAP in Passkey Authentication](#) (Kim, Shin, Ryu and Choi, IEEE Access vol. 13, 2025, DOI 10.1109/ACCESS.2025.3570377). They automated the desktop variant end to end, screen-capturing the legitimate QR out of a real browser on the attacker's PC.

QR relay latency averaged 0.785 seconds and peaked at 0.812, short enough to vanish into normal page-load time, while the QR validity window came in at a uniform **1 minute 5 seconds** across all ten production relying parties measured, namely Apple, Google, Microsoft, NVIDIA, Shopify, GitHub, PayPal, Amazon, Adobe and eBay. Against a mean human login time of 20 seconds and a slowest of 40.3, the budget closes by roughly 3x with room to spare.

The headline is narrower than it reads: the live PoC ran against a single target, NVIDIA, so **the 100% success rate belongs to that PoC alone**, and the ten-relying-party figure covers the QR validity window only, with feasibility against the other nine inferred from the timing budget rather than demonstrated. The rest of the method is in the paper.

Every defense the authors propose is QR-modal hardening, whether animated or capture-resistant QR codes or a visible countdown, and every one of them depends on the user knowing what to look for; a structural fix would bind the BLE pairing to the user's client identity, and **no such mechanism exists in CTAP 2.x**. Their threat-impact section (V.D) identifies passkey enrollment as a possible route from a relayed login to persistent access. This requires the RP to accept enrollment from the relayed session or the attacker to satisfy or relay any

additional authorization. We examine those enrollment controls under the relying party and under user and recovery. They disclosed to the FIDO Alliance before publishing.

PoisonSeed: Attempted Cross-Device Relay

Exactly one threat-actor attempt at this attack class sits on the public record, and it failed. In July 2025 the PoisonSeed crew ran an AiTM campaign against corporate Okta and Microsoft 365 tenants, with their backend picking cross-device authentication out of the identity provider's method menu so the user never chose it, and everything worked except proximity: the operators were remote, and **the BLE proximity check** is the actual limit. Expel's writeup said otherwise before [correcting it](#), so what sits on the record is a relay attempt that didn't land.

What proximity buys, then, is real protection against remote attackers, and the open question is what happens the day a threat actor pairs PoisonSeed-style campaign infrastructure with a HiPass-style co-located deployment. Righi and Langley have both published that design, a dropped BLE relay plus a phishing front end, but nobody has been observed running it and nobody ships a packaged tool for the BLE-forwarding half. Langley also [describes](#) a variant needing no spear-phishing at all, a BLE receiver hidden somewhere busy and a QR code promising free WiFi, where the receiver proxies a real authentication challenge and the transaction is, from the phone's perspective, **indistinguishable from a real sign-in**.

The FIDO: / Intent Bug (CVE-2024-9956)

A mechanistically distinct attack was independently discovered by [Tobia Righi](#), who found that Chrome, Safari and Firefox all let web pages navigate to FIDO: / URIs and so trigger Cross-Device Authentication programmatically. One click on an attacker-controlled page redirects the victim's mobile browser to a fresh FIDO: / URI, pulled live from a headless browser driving the real relying party, and a single tap plus a biometric hands the signed assertion straight back to it, with no QR code to scan and no spear-phishing email to open anywhere in the chain.

The whole thing automates, in the researcher's [CrossCheck](#), and runs on a Raspberry Pi planted near the target. All three browser families patched it by blacklisting FIDO: / from navigation by web content, Chrome first in October 2024 under CVE-2024-9956 and Firefox last in February 2025, after which the blacklist itself needed patching.

Dennis Kniep [bypassed Apple's](#) through the default-installed Shortcuts app, and iOS stayed exploitable until 18.6 shipped in July 2025, which is late enough that the class is worth keeping in mind rather than filing away as fixed.

Linked Devices

The specification calls it a state-assisted transaction and Chrome calls it a linked device: the first QR handshake leaves the desktop holding linking state for that phone, so every sign-in after it opens with a notification on the phone instead of a QR code on the screen, mechanics we saw in [The Architecture](#).

A BLE advertisement still goes out on every transaction, so the physics check survives intact, but the QR doesn't, and every attack above needs a QR code or a FIDO: / URI in front of the victim. Nobody has published an attack against linked-device cBLE specifically and we won't imply one exists. What changes is the defense side as every mitigation anyone has proposed hardens the QR modal, and a linked device never draws one, so proximity is left carrying the trust decision by itself.

What a Relying Party Can Actually Do

A signed WebAuthn assertion doesn't identify the transport used. `authenticatorAttachment` reports attachment modality, while `getTransports()` reports supported transports at registration rather than the transport used for a later authentication. Both are unsigned, client-reported values, so neither provides reliable evidence of hybrid use when the client is attacker-controlled.

WebAuthn provides no RP-enforceable option to disable hybrid transport against a hostile client. Removing the QR option or pruning `transports` affects client behavior but doesn't establish a server-enforced restriction.

To limit persistence after a relayed login, the RP should require independent step-up authorization at the passkey-enrollment endpoint. If enrollment accepts the session alone, the attacker may register a credential they control and continue authenticating after leaving BLE range. A step-up that the attacker can satisfy or relay doesn't provide that protection.

Actor 3: The Client

Who	What	How
Browser vendors, extension authors, password managers, and whatever third-party JavaScript the relying party loaded this quarter.	The broker WebAuthn assumes is honest, meaning the <code>navigator.credentials</code> API any script can call, the touch-something dialog nothing on the page can verify, and a password manager quietly answering ceremonies the browser thinks it owns.	Assume breach, then hook <code>navigator.credentials</code> , forge a whole ceremony, or hijack the assertion on its way out. Research plane , and the controls all live elsewhere.

The client handles the paperwork, with the relying party writing the challenge, the authenticator signing it, and the browser carrying both directions while it renders the dialog that asks you to touch something. WebAuthn assumes that broker is honest, which on a real machine means trusting a dozen unverified extensions and a password manager that injects its own UI into the page, along with whatever third-party JavaScript the relying party loaded this quarter and possibly a lingering XSS.

Almost all client-side passkey research is assume-breach, meaning the attacker already holds script execution in the page through a malicious extension or an XSS on the relying party's own origin. That precondition is a fair thing to push back on, because an attacker who can run JavaScript on your login page can already read what the user types, move money and open a session, so the passkey isn't what you'd be worried about first.

A hostile client sits outside what WebAuthn promises, since the spec trusts the browser to determine the origin and to broker the ceremony honestly, so nothing in this section is a break of the protocol. What's left once that assumption fails is damage control, and the private key never touching the page is meant to mean that script on that page still can't sign anything for a site it isn't on. Anything here that breaks that is worth reporting, and what you end up testing is whether the relying party, the browser vendor and the password manager still hold the line once the client stops holding it.

Every control that catches this lives somewhere else, whether that's attestation enforcement and challenge-to-session binding at the relying party or the API restriction and the top-layer UI at the browser vendor, which makes testing this actor a matter of checking whether everyone else's redundant checks hold once the client turns hostile.

Forging the Whole Ceremony, AAGUID Included

"Intercept" undersells what page-level JavaScript can do. In [Passkeys Pwned](#), Shourya Pratap Singh, Jonathan Lin and Daniel Seetoh of SquareX (2025) replaced `navigator.credentials.create` and `navigator.credentials.get` with their own functions and ran the entire ceremony inside the page, where their code mints a fresh keypair in the page's own JavaScript context, signs the relying party's challenge with it and assembles the `attestationObject` by hand with whatever AAGUID the operator picked from the extension's UI, so the demo can claim "1Password" or "iCloud" at will. The real authenticator is never asked for anything, and origin binding passes cleanly because the attacker's code is running on the legitimate origin.

Alongside the normal mode described above, silent mode is a variant where the hook calls the real authenticator as well so the victim sees a genuine Touch ID sheet and gives a genuine gesture, after which the legitimate response is thrown away and the forged one goes to the server, leaving the victim's account page showing a passkey while the server holds the attacker's public key. The force-reset option deliberately fails legitimate `get()` calls to prompt re-enrollment.

They ran it against `webauthn.io` and against a real Google account, ending in a signed-in session and a password change made from inside it, and the forged keypair went out to an attacker API as well so it could be reused from a second machine. (SquareX sells browser-security tooling, and the talk closes on a product pitch.)

A hostile extension also buys prompt fatigue, since triggering `navigator.credentials.get()` in a loop gets an approval out of muscle memory sooner or later, and the attacker times that approval to land on whatever they wanted signed.

WebAuthn does ship clickjacking protection in the `crossOrigin` and `topOrigin` fields of `clientDataJSON`, but an [at-scale audit of 103 live relying parties](#) found that 102 of them accept a manipulated value (CWE-1021, improper restriction of rendered UI layers), so the server-side backstop isn't there either.

Signed Assertion Hijacking

The clickjack most people remember from [Marek Toth's DOM-based extension clickjacking research](#) (August 2025, last updated January 2026) is one component of a longer chain, and the passkey variant is the interesting end of it. It starts with (i) getting JavaScript to run on the target's own site, because the browser will only hand a passkey to a page that belongs to the domain that passkey was registered for. The relying party picks that domain, so a passkey registered for `www.target.com` can only be used by pages on `www.target.com`, while one registered for the bare `target.com` can be used by any page below it, which puts `shop.target.com` and `files.target.com` in range.

Where the relying party made that wider choice, a stored XSS or a user-content upload on any subdomain is enough, and the relying-party section will put numbers on how often that happens.

Then (ii) you need a password manager whose prompt can be clickjacked. Most of them draw it inside the page instead of in browser chrome, and anything inside the page can be styled by the page, so the script starts the ceremony, sets the prompt to zero opacity, and drops it over a cookie banner's Accept button. The victim clicks Accept, and the click lands on the prompt they can't see.

One click signs a genuine, freshly issued challenge for the correct origin, and because the script has already repointed the submission endpoint, that assertion goes to the attacker instead of to the server. They then replay it into their own unauthenticated session, where the server sees a fresh challenge, a valid signature and the right origin, and logs them in. The flaw doing the damage sits at the relying party, because the challenge was never bound to the session it was issued to.

All 11 password managers tested were vulnerable in default configuration in April 2025, and 8 of them were vulnerable to the passkey variant specifically, across roughly 40 million installs at peak. Vendors have shipped fixes since at different speeds and with different definitions of "fixed," so pull his per-vendor matrix live rather than trusting any list frozen in a document including this one. His own conclusion is that no simple protection is available to the password managers, and that closing this properly needs a browser API that doesn't exist yet. The server-side half should worry you more, because of seven testable FIDO-certified passkey solutions he surveyed, four (Hanko, SK Telecom, NokNok, Authsignal) had no session-bound challenge.

XSS can also ride the live ceremony and defeat origin binding directly, as Catalano et al. showed in [Defeating FIDO2/CTAP2/WebAuthn using browser in the middle and reflected cross site scripting](#) (J. Computer Virology and Hacking Techniques, 2025), where the injected JavaScript runs in the authentic origin so the `navigator.credentials.get()` call passes the origin and RP-ID check cleanly while an extra HTTP channel smuggles the assertion out. It does need the relying party to host an exploitable reflected XSS, which makes it a conditional relying-party break rather than a protocol break, but the origin check is plainly no shield against an attacker who is already executing inside the origin.

The Password Manager Is a Second WebAuthn Implementation

A password-manager extension is a second WebAuthn implementation sitting in front of the browser's, running in the page and holding its own idea of which requests are legitimate, so every control you assume the browser applies is one that second implementation may never have read. Where a deployment ships browser-level restrictions as part of its defense, check whether they survive an extension being installed.

The concrete case comes from [Scott Helme](#). A relying party can switch WebAuthn off for a page with `Permissions-Policy: publickey-credentials-create=()`, `publickey-credentials-get=()`, and the browser honors that correctly, while 1Password's extension ignored it entirely. It wrapped `create` and `get` in the page's main world without ever consulting the header, brokering a ceremony the relying party had explicitly withdrawn.

The scope is narrow as 1Password closed the report as Informative and shipped hardening in v8.12.20.10 on May 15, 2026, and the adjacent "approve an attacker's keypair" step was never a 1Password bug at all, so the ignored header is the whole of it.

What Actually Defends This

Where the RP requires verified authenticator provenance, it must verify the returned attestation statement and assess its type and trust path against its policy. `indirect` can provide verifiable attestation, including through an Anonymization CA; requesting `direct` or `enterprise` alone doesn't establish trust. The FIDO Metadata Service or another trusted source can supply trust anchors and authenticator metadata. `fmt: none` and self-attestation don't independently prove the claimed authenticator model.

An unverified AAGUID can support inventory or a risk heuristic, but an attacker can copy a familiar value. It cannot establish provenance or justify bypassing an enrollment step-up. Google's strict CSP and Trusted Types made the SquareX team's Gmail PoC harder to build by their own account, raising the cost of getting script into the page without closing the class.

Against the assertion-hijack chain, the session-bound challenge at the relying party is the fix, and its best property is independence, because it holds whether or not the victim's password manager ever ships a patch.

Platform-native passkey UI (Windows Hello, the macOS Touch ID sheet, Android Credential Manager) and hardware FIDO2 keys defeat the DOM-clickjack component outright, since that UI is drawn by the OS outside the page's DOM and page script can't set its opacity to zero.

Both are server-side or platform answers to a client-side attack, and both are testable from a proxy in an afternoon by assuming script execution on the origin and then asking whether the server will accept an `fmt: none` credential that page JavaScript minted itself, and whether it will accept a valid assertion from whatever session submits it.

Actor 4: The Relying Party

Who	What	How
The team that implemented FIDO2 support: their own code, the WebAuthn library they picked, and the product decisions around it.	Every server-side verification step, plus registration, enrollment, credential management and everything the spec leaves to the implementer.	Decode the ceremony, tamper with a field, re-sign, replay, and see what the server accepts. Engagement plane , and the only actor here you can fully cover inside a scope.

The relying party validates WebAuthn registration and authentication responses before accepting a new credential or granting access to an account. Errors or omissions in those checks can permit unauthorized registration or authentication even when the authenticator and the underlying cryptographic protocols operate correctly.

Researchers at Ruhr University Bochum and their collaborators' [The State of Passkeys: Studying the Adoption and Security of Passkeys on the Web](#) (Jannett et al., USENIX Security 2026) ran, as far as we know, the first at-scale active security audit of live passkey relying parties, where earlier empirical work was small-sample or assumed a compromised client rather than a remote web attacker, and their PASSKEYS-ATTACKER tooling emulates both the browser client and the authenticator so it can tamper every WebAuthn field on the way past.

Of the 872 passkey-enabled sites they discovered, 208 could be registered on and 103 fully security-tested, and not one of the 103 passed every check the standard mandates, with 18 carrying critical-CVSS findings (account takeover, passkey deletion, lockout) and 53 carrying high-CVSS. Any hard count attributed to "the audit" anywhere in what follows comes from there.

The sample has a bias worth knowing about, since it skews toward sites that let you register a passkey with no extra verification, which may systematically be the weaker ones, and sectors like online banking couldn't be tested at all, so "all of them" means all of that sample rather than every passkey relying party on the web.

We sit as a proxy between the client and the relying party, and from there we trigger both ceremonies, take every field the client forwarded, change it, and watch whether the server notices and what it does when it does.

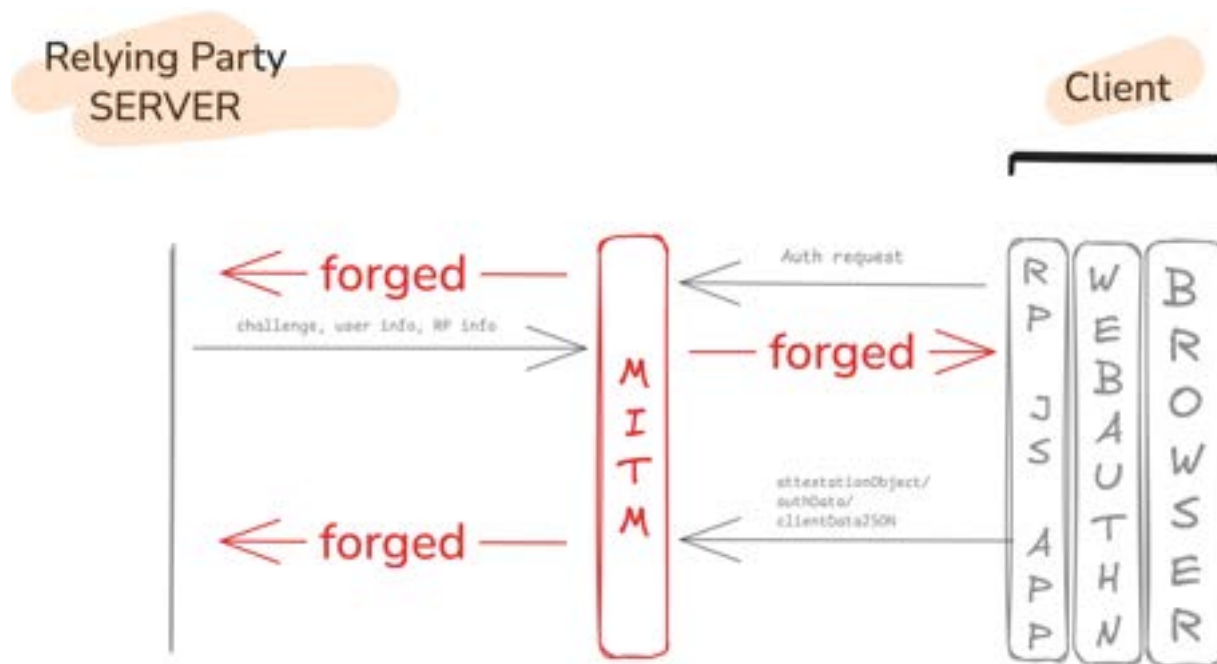


Figure 5: WebAuthn Traffic Tampering

Everything in this section is a variation on that one move, because WebAuthn hands the server roughly 22 verification steps and the only way to learn which of them the deployment team implemented is to break each one in turn and see whether anybody complains.

Several independent researchers have [explored](#) this surface and keep landing on the same conclusion, that the protocol hands the relying party everything it needs to make a secure decision and most relying parties don't use it.

Two practical problems sit in front of everything else, since you can't read passkey traffic in Burp Suite, and once you can read it you still can't put it back on the wire after changing a byte, which is the reason this surface stays untested on most engagements and why both come before the field-by-field walk.

Decoding and Re-encoding WebAuthn Traffic

Burp Suite shows you a Base64 blob, and everything you want to change sits three layers underneath it. The ceremony ships CBOR, a binary serialization Burp Suite has no native view for, and relying parties almost never put raw CBOR on the wire, since they Base64 it first and frequently twice, mixing url-safe and standard alphabets and padded and unpadded variants inside the same request, after which plenty of them wrap the result in a vendor JSON envelope with field names they invented.

The specifications standardized the ceremony and stopped there, because nothing in WebAuthn says what the HTTPS request carrying that ceremony has to look like, so every deployment filled the blank on its own.

A large share of them never fill it themselves, since [Bhardwaj and Sastry's scan of the Tranco Top 100K](#) (PAM

2026) found 75.2% of passkey-supporting sites reaching passkeys through Google SSO and only 18.6% using a JS library you could fingerprint, so you can't assume a library shape and you can't assume a wrapper.

Microsoft, GitHub and Google run the same ceremony and put the same two fields, `clientDataJSON` and `attestationObject`, on the wire three completely different ways, in traffic we captured while preparing the DEF CON 34 talk.

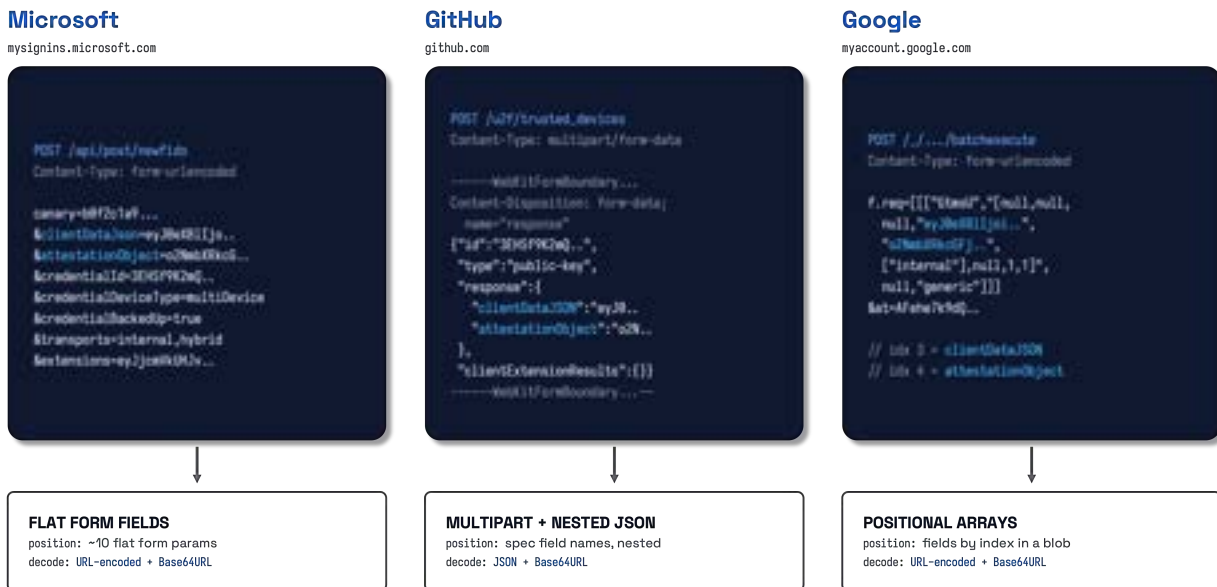


Figure 6: Three Vendors, Three Wrappers

Microsoft explodes the response into flat form fields on `/api/post/newfid` with no nesting and no JSON object to grab, GitHub keeps a JSON object but buries it in a multipart form on `/u2f/trusted_devices`, and Google hides the same fields inside a stringified positional array in its `batchexecute` RPC, where there are no field names at all, only relative positions inside nested arrays, and then URL-encodes the result.

All three decode to the same fields, so the ceremony underneath is identical and the spec is doing its job. Once the wrappers come off, this is what you're tampering with:

```
{
  "clientDataJSON": {
    "type": "webauthn.get",
    "challenge":
      ↪ "zYJx-8mHw8wK7vC4qRseSJrDCd01yKI fZk_njXE0oeuQD7CuKUoQ2frvV0NB0JiVZSBgjUYy8vGb-0Lq-BS1wA",
    "origin": "https://webauthn.io",
    "crossOrigin": false
  },
  "authenticatorData": {
    "rpIdHash": "74A6EA9213C99C2F74B22492B320CF40262A94C1A950A0397F29250B60841EF0",
    "flags": {
      "userPresent": true,
      "userVerified": true,

```

```
    "backupEligible": false,  
    "backupState": false,  
    "attestedCredentialData": false,  
    "extensionDataIncluded": false  
  },  
  "signCount": 42  
},  
"signature": "304402207BC3E1F0A2D4C6980B5E3F1A2C4D6E8F0A1B2C3D4E5F60...DDEEFF02",  
"userHandle": "6D617474656F2D67696F7264616E6F"  
}
```

Manual modification requires decoding the wrappers and CBOR, then preserving each layer's Base64 variant and padding when reconstructing the request. The challenge may expire during that work.

That gap is one of the reasons why [Passkey Editor](#) exists, unwrapping the chain layer by layer while remembering the exact Base64 flavor and padding of each one so it can put them back unchanged, and re-signing with a key it plants at registration, which is what turns every field walk below from a thought experiment into a request you can send. For more details please see the companion post, [Introducing Passkey Editor](#).

That walk starts here, taking the values an attacker can actually reach and asking what changing each one buys. We take the structures themselves as read, since [The Protocol](#) carries the authData layout and the flags table, and [The Architecture](#) the machinery built on top of them.

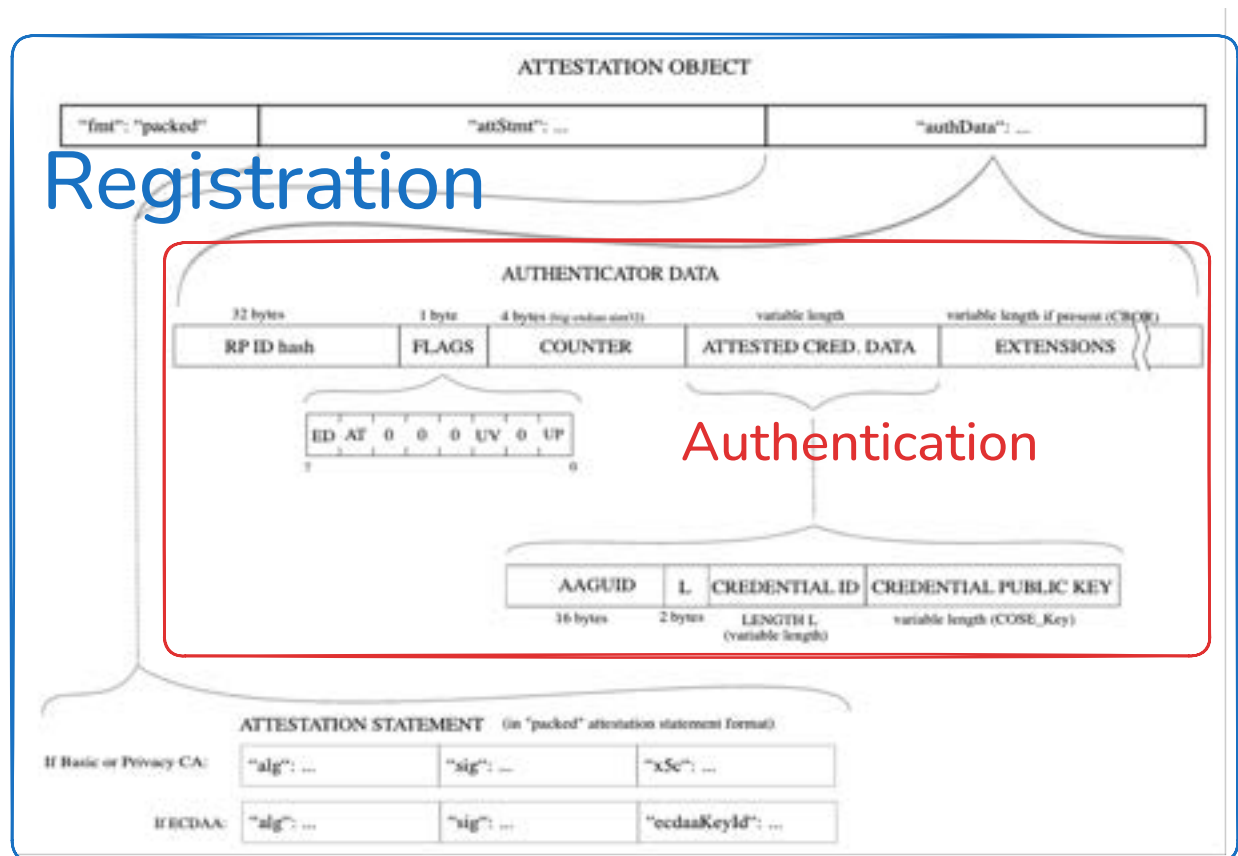


Figure 7: Attacker-Controllable Ceremony Fields

Origin and rpIdHash Validation

The `clientDataJSON` carries the `origin` among its other fields, and the relying party must verify it matches its own domain.

The `rpIdHash` in `authenticatorData` says which relying party a credential belongs to, and the `origin` in `clientDataJSON` says where the ceremony actually ran, which is why both get checked and why the second one catches what the first can't. No browser will let `evil-bank.com` touch a `bank.com` credential, because an origin may only claim its own effective domain or a registrable suffix of it, so the authenticator is never even asked.

It will happily let `dev.bank.com` ask for `bank.com`, though, so wherever a credential was scoped to the registrable domain rather than to the host that needs it, a subdomain takeover or a stored XSS under that domain yields a real assertion, signed by the victim's own authenticator and carrying the correct `rpIdHash`, where `origin` is the only field that gives it away. Relying parties that never read it accept that assertion, and those that do read it still get it wrong through patterns like `/. *bank\.com$/` or by accepting any subdomain. The key point here lies in the allowlist behind all the checks.

That vector has hard numbers now, because the audit found origin and `rpIdHash` validation failures on 40 of 103

relying parties (CWE-346, CVSS 8.1), which re-enables phishing against supposedly phishing-resistant passkeys, with the origin accepted cross-site on 7 sites, on a subdomain on 23 and on the parent domain on 31, while rpIdHash was accepted cross-site on 8, on a subdomain on 7 and on the parent on 5. Upscoping is its own problem, since 80 sites, 38% of the 208 registrable relying parties, scope passkeys to the registrable domain or TLD instead of the specific subdomain that needs them, which is the `dev.bank.com` case measured in the wild and the reason Squarcina et al.'s "Can I Take Your Subdomain?" work on dangling subdomains applies directly here.

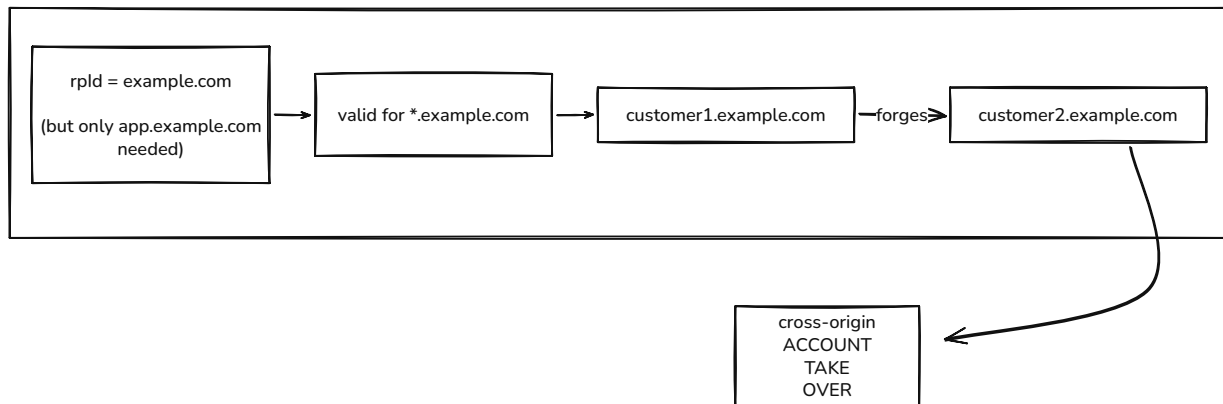


Figure 8: RP-ID Over-Scoping

The permissive-allowlist pattern shows up in real code rather than only in regex hypotheticals. Maarten Balliauw's Duende writeup (*Relying Party ID and Origin With Passkeys*, October 2025) gives you C# quotable almost verbatim in findings, where a server check like `Origin.StartsWith("https://") && Origin.EndsWith("example.com")` lets an attacker-controlled `customer1.example.com` use or forge passkeys valid for another tenant's `customer2.example.com`, a clean cross-tenant theft.

Scott Helme fixed the same shape while open-sourcing *passkeys-php* (May 2026), an `rpId` match treated as a substring suffix so `example.com` matched the host `evil-example.com`, and his writeup lists the other checks worth stealing for a test plan, starting with requiring `fmt: none` to carry an empty CBOR map. He deleted roughly 1,100 lines of attestation handling on the way while keeping only that one, which tells you how much of that surface real libraries carry and rarely test.

All of that has a named, patched instance, since the PHP *web-auth/webauthn-lib* ran both the configured `allowed_origins` and the incoming `clientDataJSON.origin` through `parse_url` and compared host components only, discarding scheme and port, so an origin differing solely in scheme or port passed validation where the spec says to compare byte-exact (CWE-346, CVSS 5.4, affecting 5.2.0 up to 5.2.4 and fixed there on March 10, 2026).

Related Origins and Dangling Domains

Relying parties wanting to share passkeys across several domains publish a `/well-known/webauthn` allowlist, which WebAuthn Level 3 calls Related Origin Requests. The audit measured 2,177 related origins across 177 domains and found one allowlisted domain still registrable for 10 EUR per year, where acquiring that single

dangling domain would grant passkey access on 72 related origins ([CWE-610](#), CVSS 8.8). The origin allowlist is a maintenance liability where one expired entry becomes a key to dozens of relying parties.

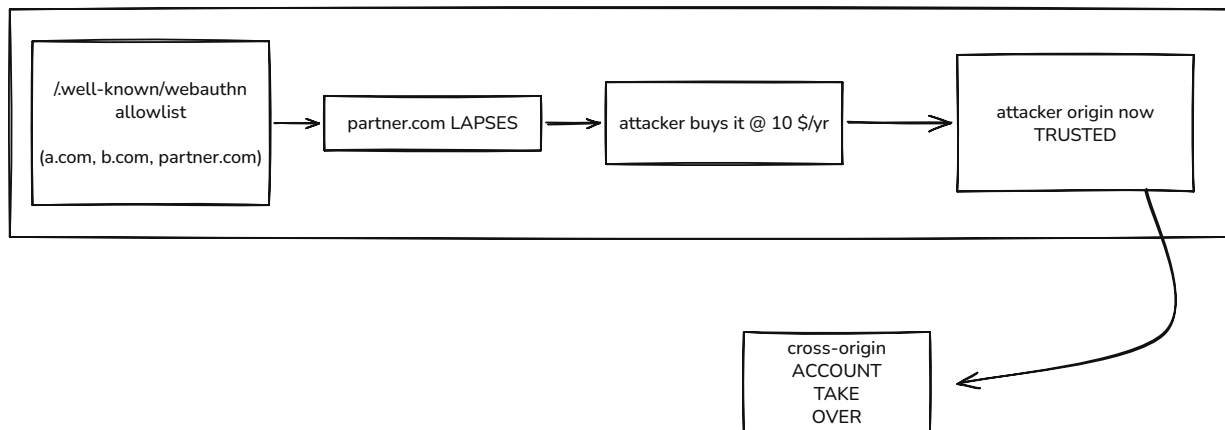


Figure 9: Dangling Domain Takeover

The Flags Byte

Two bits in the authData flags byte carry the whole story here: UP, user present, meaning somebody touched the thing, and UV, user verified, meaning a PIN or a biometric was checked. [The Protocol](#) has the full bit table and two worked decodings if you want the layout.

`userVerification: "required"` requires a conforming client/authenticator to satisfy user verification or fail the operation. To test the RP's enforcement, keep its server-side policy unchanged and produce a correctly signed `UV=0` assertion using an enrolled controlled authenticator, or change the client-side request to `discouraged` before calling `navigator.credentials.get()`. Changing the request doesn't guarantee `UV=0`: authenticator or credential policy may still require verification.

Changing a signed flag after the response is produced invalidates the signature, so rejection of that response doesn't isolate UV-policy enforcement. Test signature enforcement by corrupting only the signature of an otherwise policy-compliant assertion. Reading the flag before signature verification isn't itself a defect if a failed signature still prevents acceptance. We've seen backends accept correctly signed `UV=0` responses despite requiring verification because the developer enforced the requirement only in the frontend.

The audit turns that anecdote into counts, and they're worse than the anecdote suggests, since `UV=false` was accepted by 10 relying parties despite a `UV: required` policy and `UP=false` by 27, which turns the authenticator into a silent signing oracle that signs without even asserting a human was present. Worse, 5 of 103 skip signature verification entirely and accept a last-bit-flipped signature for a confirmed account takeover ([CWE-347](#), CVSS 9.8), while 12 more show create-versus-get signature and context confusion, failing to separate registration assertions from authentication ones.

The bit-flip you were going to try by hand is a measured bypass on a non-trivial slice of production relying parties.

COSE Algorithm Downgrade and Confusion

The relying party publishes acceptable COSE algorithms in `pubKeyCredParams` at registration, and in the same spirit as a TLS downgrade check that list is a stated preference until somebody proves the server enforces it on the way back in, so the test worth running is registering with an algorithm the list never offered and seeing whether it's taken. The audit's algorithm hygiene data says the surface is real, with Jannett et al. counting 26 deprecated or unassigned algorithm IDs being requested, RSASSA-PKCS1-v1_5 with SHA-1 five times and symmetric HMAC eight times.

Those numbers describe what relying parties *ask for* in `pubKeyCredParams`, which makes them hygiene and library fingerprinting rather than a measured count of servers that accepted a downgraded signature, and nobody has published that count as far as we know.

What the numbers establish is that the check is worth running, not what happens when you run it. One datum from the same section is pure recon value, since 22 sites omit only ES384 from their list, a shape distinctive enough to fingerprint the underlying library before you've sent a single tampered request.

Challenge Freshness and Session Binding

A common flaw is failing to invalidate the challenge immediately, and on assessments we've captured a valid login response, `authenticatorData` plus signature, and replayed it minutes later, because the backend treated the challenge as a session ID with a long timeout rather than a number used once. An attacker who obtains the complete authentication response through a compromised client, a TLS-terminating proxy, or server or SIEM logs that retain request bodies may be able to replay it. Successful replay requires the RP to accept the previously used challenge and the request to satisfy or bypass its session-binding checks.

The audit gives this hard counts too, with challenge handling broken on 22 of 103 relying parties ([CWE-384](#), CVSS 8.2), covering challenge reuse, cross-session acceptance where a challenge issued in one session is accepted in another (9 at registration, 14 at authentication), and bit-flip, so roughly one in five live relying parties mishandles the most basic freshness guarantee in the protocol.

Registering a Key on an Account You Don't Own

During registration the server must ensure the user initiating `navigator.credentials.create()` is the authenticated owner of the account, and where the endpoint lacks session binding an attacker can start a registration ceremony for a victim's account by submitting their email to the registration API, receive a `PublicKeyCredentialCreationOptions` object tied to the victim's user ID, complete the ceremony with their own authenticator, and end up with their own public key bound to the victim's account.

A cloud identity provider reaches the same ending with no browser involved at all, since an attacker holding Entra ID's `UserAuthenticationMethod.ReadWrite.All` Graph permission can [register their own FIDO2 key on any account up to Global Administrator](#), which is the passkey version of the permission [SpecterOps abused against Temporary Access Pass](#), so treat it as Global-Administrator-equivalent in any tenant with passwordless enabled.

A variant that never starts the ceremony is worth knowing mainly because it stopped working. [Peizhou Chen's MSc thesis](#) (University of Twente with Computest, March 2024) tried to forge registration cross-site so the victim's

own session would bind the attacker's key, which needs three failures at once, a session cookie set to `SameSite=None`, no session-tied CSRF token, and a challenge the server never bound to the session. None of Chen's targets got all three wrong, and his one success was `webauthn.io`, a demo that keeps no cookie session at all, where he notes its `SameSite=Lax` cookies would have stopped him had there been one. Chrome has defaulted to `Lax` since 2020, so what survives is the structural point rather than the attack, that the WebAuthn challenge doubles as a CSRF token when the relying party ties it to the session, and the spec never says it has to.

Registering a new passkey should be gated on an existing strong factor, because it's the moment an account grows a new durable way in and deserves at least as much friction as a password change. The audit measured how often that happens: of 208 relying parties where a passkey could be registered, only 71 (34%) required any extra confirmation at all, and where a confirmation did exist it usually fell back to something phishable anyway, password on 37 sites, email on 25, SMS on 6.

Two thirds of registrable relying parties will let an attacker with a foothold add a credential that looks phishing-resistant, is durable, and typically survives a password reset, with no friction at all, while a third of the remainder gate it behind a factor an AiTM proxy already defeated on the way in. This is the single cheapest persistence primitive in a passkey deployment.

Credential IDs and the Overwrite Class

The credential ID is the handle the authenticator mints at registration to name a key pair, supplied in `allowCredentials` for non-discoverable authentication so the authenticator knows which key to sign with and used server-side as the lookup key that maps an arriving assertion to a stored public key and its owner, which *The Protocol* walks byte by byte.

Where the backend doesn't enforce global uniqueness for credential IDs, an attacker can register a credential whose ID collides with a victim's existing one, and depending on the storage logic that overwrites the victim's credential and locks them out, a targeted denial of service against one account's passwordless login. That used to be where the story ended.

The audit promotes it to a measured family it calls **Credential Overwrite** (CWE-639, CVSS 9.1) with outcomes worse than lockout, rooted in the relying party failing to bind a credential ID to exactly one user, so an attacker registers a passkey reusing the victim's public, unauthenticated `credId`. Table 2 reports 14 distinct affected relying parties among the 103 tested. The detection-method counts overlap and must not be added together.

Registering the victim's credential ID with the attacker's public key was accepted by 12 RPs; the reported outcomes included credential replacement, conflicting records and credential deletion. Registering both the victim's credential ID and public key under the attacker's account was accepted by 13 RPs. The authors demonstrated resulting session swapping on three sites: the victim authenticated into the attacker's account, potentially exposing data subsequently entered there. This variant requires access to the victim's public key as well as the credential ID. Public-key-only substitution was accepted by two RPs.

Attestation Policy and Library Verification Failures

At registration, the `attestationObject` can carry evidence of authenticator provenance that the RP must verify and assess against its trust policy before using it to enforce model restrictions. *The Protocol* has the `formats` field by field.

Actor 1 left two questions for the relying party, and the first of them, whether the relying party enforces attestation and the AAGUID allowlist when it claims to, is answered by named CVEs. Two recent Keycloak advisories show what happens when it doesn't, since [CVE-2025-12150](#) bypasses a configured DIRECT-attestation policy with an attestation object carrying `fmt : none` ([CWE-347](#), fixed in 26.4.4), while [CVE-2026-6856](#) (Sylvain Pelissier, SICPA) is subtler, because packed self-attestation with no x5c chain leaves the AAGUID unverified and the registration accepted, defeating the exact allowlist meant to mandate specific authenticator models.

One layer down, in the libraries relying parties trust to do the verifying for them, `return true` stops being a figure of speech, and three CVEs reach that outcome by three different routes, every one of them ending with the server handing out a session for the wrong user.

- [CVE-2025-26788](#) in StrongKey FIDO Server (CVSS 8.4, [CWE-639](#)) treats a non-discoverable flow as discoverable without binding the assertion to the initiating username, so an attacker starts authentication as the victim, substitutes their own credential ID and key, and is logged in as them.
- [CVE-2026-46419](#) (YSA-2026-02) in Yubico java-webauthn-server, the canonical reference library, had `finishAssertion` return success for a credential owned by a different user than the one named in `StartAssertionOptions.username`, affecting 2.8.0 to 2.8.1 and fixed in 2.8.2 and 2.9.0, with impact bounded to accounts not yet enrolled in WebAuthn.
- [CVE-2024-12225](#) in Quarkus quarkus-security-webauthn (CVSS 9.1, [CWE-288](#)) leaves the framework's default register and login endpoints reachable when a developer adds custom ones, which hands an attacker a login cookie for any user whose name they know, and the root cause is leftover defaults rather than an origin bug.

Signature Counters and RP Policy

The Architecture introduced `signCount` as a potential cloning signal. Authenticators that implement signature counters increase them; those that don't keep them at zero. [U-Zyn Chua](#) reported in May 2025 that Apple, Google and Microsoft returned zero for synced passkeys. Jannett et al. reported that more than half of the tested RPs ignored the counter, while the rest blocked out-of-sync logins.

WebAuthn calls for a counter comparison when either the stored or received value is nonzero. A received value less than or equal to the stored value is a risk signal that may also result from an authenticator malfunction or out-of-order processing. The specification leaves rejection and counter updates in this case to the RP. Keeping `max(stored, received)`, as Chua recommends, preserves a high-water mark; it doesn't require automatic rejection of lower values.

The fast-forward scenario described in *The Architecture* depends on two conditions: the attacker can produce a validly signed assertion with an inflated counter for the targeted credential, and the RP stores that value and subsequently rejects legitimate assertions with lower or equal counters. Because `signCount` is covered by the signature, producing that assertion requires the credential's private key or control of an authenticator capable of signing the chosen counter value.

We tested this fast-forward behavior separately from the cited audit. RPs can retain a high-water mark and use non-increasing counters for logging or risk scoring without automatically rejecting authentication. Acceptance of a non-increasing counter alone therefore doesn't establish that the RP ignored it.

Checklist and Creativity

That's a long list of things to test against one server, and it's worth sorting before you plan your time, because it splits into two kinds of work that ask completely different things of you.

On one side is a reusable set of protocol-verification checks that tooling can help cover efficiently, including signature validation, credential ownership, origin checks, challenge handling, and backup-flag consistency. The applicable checks and expected outcomes depend on the protocol version and deployment policy. The [test-case reference](#) provides a starting point, with the audit's measurements where available.

On the other side is the part no dropdown will ever find, because it's where the relying party made product decisions rather than protocol ones, which means asking how they scoped the RP ID and whether anything dangles in `.well-known/webauthn`, whether registration can be framed or clickjacked or reached by CSRF or done through an over-granted admin API in a cloud tenant, and whether enrollment requires a step-up at all. The same lens covers what persistence looks like after a compromise, what the recovery flow downgrades to, whether mixed-mode fallback is still alive, what the application binds a session to once the ceremony is over, who is allowed to delete a credential from the credential-management endpoint, and which default endpoints somebody forgot to delete. This is RP-by-RP.

Use tooling for repeatable protocol checks, then assess the deployment-specific decisions described above.

Actor 5: The Sync Fabric

Who	What	How
First-party (i.e., Apple or Google) sync, or a third-party password manager holding the credential on their behalf.	The cloud account that guards the passkey, the vault crypto underneath it, and the BE/BS signal relying parties read without validating.	Take the account rather than the ceremony, through phishing, a SIM swap or enrolling a device the fabric already trusts. Engagement plane , on its red-team half, and inside a normal scope what you test is the relying party's side of the boundary.

Synced passkeys extend the security boundary to the provider account and its authentication and recovery controls. Apple and Google both default to a device-bound push prompt rather than SMS and both now support a passkey on the account itself, but phone-number recovery stays reachable on most of these accounts, opening to other fallback ladders.

Nothing in this section breaks WebAuthn itself: the ceremony runs clean, the origin check holds, the signature verifies, and the attacker simply arranges to be a device the sync fabric already trusts, so the ceremony runs correctly on their laptop instead of the victim's. That makes it red-team territory rather than two-week-assessment territory, because the work is account takeover and device enrollment, and the passkey is the loot rather than the target.

[The Architecture](#) drew the line between device-bound and synced passkeys. However, to be more precise, two additional categories should be taken into account. Büttner and Gruschka's taxonomy ([Device-Bound vs. Synced Credentials](#), ICISSP 2025) splits passkey storage four ways, and each step outward hands the credential to one more party the user doesn't control.

1. A **device-bound** credential never leaves the hardware you carry, so the trust stops at the chip in your hand.
2. Syncing moves the boundary out to a **cloud** account, and from that point whatever guards the account guards the passkey.
3. A **shared** vault moves it out again to every member holding it, which makes the weakest of them the credential's real security level.
4. **Export** through the [Credential Exchange Format and Protocol](#) takes it out of the provider altogether. Designed to be smooth and zero-knowledge transfer, but still, it's an expansion of the attack surface.

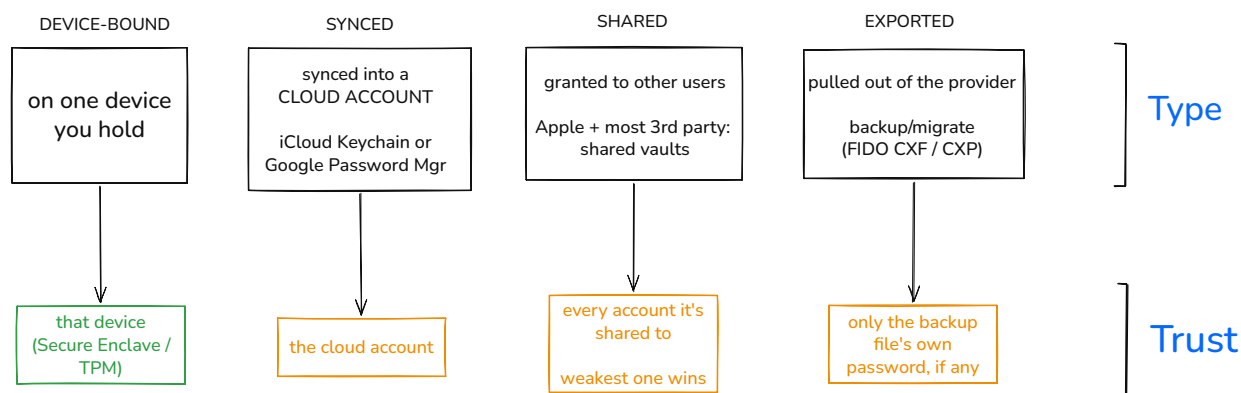


Figure 10: Four Passkey Storage Models

Trust Dependencies in Synced Passkeys

The same authors score those four types against the 25 usability, deployability and security benefits from Bonneau et al.'s 2012 password-replacement [framework](#), and the one synced passkeys lose outright is **No-Trusted-Third-Party**, because a provider now holds the encrypted material and owns the enrollment logic deciding which devices get a copy. You can trust that provider quite a lot, and in Apple's and Google's case there are decent engineering reasons to, but the **property** itself is gone.

Cambridge's [SoK on web authentication and recovery](#) (Blessing, Hugenroth, Anderson, Beresford, PoPETs 2025) doesn't hedge it, finding that syncing "inherently increases the attack surface, leaving credentials vulnerable to the compromise of the weakest synced device," and that "the practical security of synced credentials reduces to the security of the cloud account, including cloud provider security." More often than not, you can see cases where somebody's corporate passkeys turn out to live in a personal Google account a SIM swap can take.

First-party and third-party sync are different architectures, and which one is in front of you decides which attacks below apply. Apple and Google own the hardware, the OS and the sync servers, and the Cambridge SoK singles out iCloud Keychain as the one synced design reaching "a similar security level as device-bound credentials," because key material moves **enclave to enclave** via Apple-owned HSM clusters and never leaves hardware storage. A third-party manager can't match that and stay portable across ecosystems, so at some point the key decrypts into ordinary application memory on a machine the provider doesn't build.

The same split shows up at enrollment, where iCloud Keychain asks an already-trusted device to approve a new joiner while Google's security-domain join takes the six-digit GPM PIN or a screen lock with no cross-device approval: one design puts a device you're already holding in the loop, the other asks for a number a convincing caller can talk you into reading out loud. That difference is narrower than it looks, though, because the architecture buys a defense against a breached sync backend and nothing at all against a cloud account lost to a SIM swap or a well-timed phish, which is the far more common way in.

Three Routes Into the Sync Account

The classic route is **account takeover**, which Prove has [documented in the wild](#). The attacker acquires the victim's cloud sign-in credentials through phishing, reuse or an infostealer, then performs a SIM swap to

intercept the phone number and satisfy the account's SMS-based 2FA before signing in on a device they control. Getting the passkeys to follow takes one more step, because the ecosystem holds a separate keychain-release check, Apple's escrow path wanting the account password, an SMS code and a device passcode while Google's wants the GPM PIN or a screen lock, and once that's satisfied the cloud syncs the victim's passkeys to the attacker's hardware. Nothing breaks on the victim's end, and the attacker holds a cryptographically valid clone of their authentication material.

The second route skips the SIM swap entirely, which makes it the cheapest of the three, and what it turns on is **human emulation**. Chad Spensky, at DEF CON 33, phishes the sync-account credentials the ordinary way, then uses that access to drive a real browser as the victim with keyboard and mouse input rather than script. That's the detail doing the work, because the WebAuthn protections that stop a scripted attacker walking a credential out of the browser never engage when, as far as the browser can tell, a human is clicking.

Demonstrated end to end against Chrome's password manager and Bitwarden, in the [PoC](#) and [talk](#) (Spensky, 2025), and while he argues it generalizes to any synced store, iCloud included, only those two are demonstrated, so the generalization is his rather than the experiment's. He's also affiliated with Allthenticate, which builds device-bound authentication. Main takehome message here is that **a stolen synced passkey is frequently worse for the victim than a stolen password**, because plenty of relying parties drop their remaining MFA layers once a passkey is enrolled and some drop password fallback entirely, so the credential sold as the strong factor ends up the only factor and the attacker has a copy of it.

The third route goes at the **fabric** directly, since in May 2026 Curtis Brazzell of PhishU published [VaultJacking](#), which he reports testing against live Google accounts, and the chain is uncomfortably short, because a single captured 6-digit Google Password Manager PIN releases the Security Domain Secret, after which a containerized Windows VM running a virtual TPM joins the victim's security domain and the vault drains. The Apple contrast is the whole attack, since iCloud Keychain prompts an already-trusted device to approve any new joiner and Google didn't gate the PIN that way.

Vault Cryptography and Backup-State Validation

The deeper version of "synced passkey safety equals cloud safety" is that the **cloud crypto can be wrong**. An ETH Zurich and USI team ([Scarlata, Torrisi, Backendal, Paterson, Zero Knowledge \(About\) Encryption](#), USENIX Security 2026) analyzed Bitwarden, LastPass, Dashlane and 1Password under a fully-malicious-server model and found 27 working attacks (Bitwarden 12, LastPass 7, Dashlane 6, 1Password 2), up to complete compromise of all organization vaults. All four are third-party and the first-party stacks weren't in scope, which leaves them untested rather than clean, and where a threat model puts a synced passkey in one of these vaults, "the provider can't read it" deserves scrutiny.

The BE and BS flags are the protocol's own detection mechanism for synced credentials, and most relying parties read that signal without validating it. The audit found they largely fail to check BE/BS consistency between registration and authentication, whether registering a credential as device-bound then authenticating as backup-eligible, or sending the one combination the spec rules out, BE=0 with BS=1, since a credential that was never eligible for backup can't currently be backed up, with roughly 55 to 65 of the 103 tested relying parties failing the sub-tests. The security impact depends on the RP's policies and how it uses the backup-state flags; some cases may have no impact.

Compromise Detection and Emerging Defenses

What makes this actor worth a red team's time is **what doesn't happen afterwards**, because passkeys get treated as the strong factor and a session opened with one frequently doesn't trip the risk engine that the same session opened with a password would.

Persistence bought through the sync fabric then stays quiet for as long as the attacker wants it to, which [Daffalla et al.](#) measured rather than assumed. Those counts belong under the user, and that's where they are.

Defenders are moving here too, and the published work splits into two groups worth keeping apart, one that assumes the credential will eventually be stolen and tries to notice, and one that tries to make the stolen copy worthless.

- **Detection.** [CASPER](#) plants decoy passkeys an attacker can't tell apart from the real ones. Somebody spending a drained vault trips a decoy, and the relying party finds out it has been breached, which catches the theft without doing anything to prevent it.
- **Prevention.** [Bicakci et al.](#) sync ciphertext only, and keep the decryption key on a hardware token the user carries, so the cloud never holds anything usable on its own. The price is that the user has to carry the token, which is the exact friction synced passkeys were invented to remove. It's an unrefereed January 2026 preprint, so what's on offer is an architecture rather than a result.

What Falls Inside a Normal Scope

Nobody on a scoped application test is going to compromise iCloud Keychain, and it isn't worth spending the time trying. The reason this actor still matters is that it belongs in the threat model, because attackers reach for it early and often, given that phishing the provider account costs less than phishing the relying party, the tradecraft is the tradecraft they already have, and the telemetry works in their favor for the reasons above.

There's plenty you can still do inside a normal scope, mostly on the relying party's side of the boundary, asking what guards the cloud account that guards the passkey and checking whether the server rejects an inconsistent BE/BS pair wherever a policy decision rests on backup state.

Actor 6: The User and Recovery

Who	What	How
The person holding the credential, the helpdesk behind them, whoever decided the fallback had to stay, and sometimes somebody who shares their home.	Every path back in when the passkey isn't available (reset links, SMS codes, lost-device flows, support channels, alternate grants), plus enrollment, revocation, and whether anybody gets told.	Ask for the downgrade and take whatever the account hands over instead, or, with the unlocked device in reach, enroll your own biometric and your own passkey. Both planes , and the cheapest thing on this map by a distance.

Every actor examined so far has presupposed an adversary who defeats a control that was meant to hold, whether that control is implemented in silicon, in the cross-device tunnel, in the browser, in the relying party's verification logic or in the sync provider's vault. This final actor inverts that assumption entirely, since none of its attack paths require anything to fail.

What they require instead is that the deployment continue to function as designed, because a credential bound to a single device is unusable the moment that device is lost, damaged, replaced or left behind at a former employer, and a system offering no remedy for those cases is a system that strands its own users. Every deployment therefore provisions an alternative route to the account, and it's that route, rather than any weakness in the ceremony, that the attacks below take. An adversary who enters through it never interacts with the passkey at all.

According to the [FIDO Alliance's State of Passkeys 2026](#), about 30% of organizations run passkeys as the primary workforce sign-in while roughly 57% still lean mainly on passwords or other phishable methods, and the share that has gone fully passwordless, with every weaker method removed, is smaller still. In practice a passkey sits next to a pile of SMS codes, email reset links, security questions, lost-device flows, a helpdesk with a phone number, and an OAuth device-code grant nobody remembered to scope.

Every other stop on this map costs a researcher weeks or an operator a foothold, while this one, ironically, could cost a phone call. On a red-team campaign it's usually the first thing anyone looks for and the last thing that gets fixed, because killing the fallback means accepting that somebody will eventually lock themselves out for good.

A workforce tenant has a helpdesk, a badge, a manager who can vouch for an employee and an IT team that can re-enroll a laptop the same afternoon, so it can realistically enforce strict passwordless with no phishable fallback at all and absorb the support cost of doing it. A consumer service has none of that, and the recovery flow is the only thing standing between a locked-out user and an account that's gone for good, so the same recommendation that reads as routine hardening for an enterprise turns into a product decision about how many customers you're willing to lose.

The Mixed-Mode Problem

Deploying passkeys merely as an alternative sign-in method alongside phishable credentials provides UX value and **does not deliver the security promise that passkeys make**, because passkeys themselves are phishing-resistant while an account supporting active phishable fallbacks is not.

The north star is **accounts** that are phishing-resistant rather than **methods**, and more factors don't get you there, because not all steps are factors and the marginal value of keeping a password on top of a passkey is **minimal to negative**, since retaining it just in case retains the vulnerability to credential stuffing and AiTM. **Adding phishable steps lowers the security ceiling of the entire account to the level of the weakest credential.**

The UK NCSC describes these risks in its April 2026 paper [Comparing the security properties of traditional user credentials and FIDO2 credentials for personal use](#): recovery "can often be weaker than the authentication," email-based recovery is "effectively reducing the security of the service to that of the email account," and the recommendation is to eliminate fallbacks entirely, since "these attacks should be prevented if only passkeys are used, with no weaker password or tMFA options supported as a 'fallback'."

The deployment figures above make fallback and recovery controls a priority for assessment. Teams often retain passwords, SMS OTP and magic-link recovery to preserve usability and reduce account lockout.

The Downgrade Path

Auditing a passwordless implementation starts by looking for the **"I can't use my passkey"** link and seeing where it lands you, which on nearly every engagement is one of six paths: whether that's SMS one-time codes, email reset links, security questions, lost-device flows, the helpdesk or an OAuth device-code grant.

Proofpoint has [documented a practical downgrade](#) against Microsoft Entra ID, in which a phishing proxy spoofs an unsupported browser User-Agent, causing Entra to disable passkeys and offer SMS or OTP in their place. The victim is presented with a dialog identical to the one they have always seen, minus the single row that would have protected them.



Figure 11: Method Selection, Before



Figure 12: Method Selection, After

Recovery flows deserve the same push, because account recovery often becomes the de-facto authentication mechanism for all the people forgetting about their password, so if you can compromise the email you can

usually decouple the user's security key, and in magic-link systems the inbox *is* the master key, giving both direct login and the ability to enroll a passkey persistently, as we walked in [The Architecture](#). Netcraft ([Spicer, April 2026](#)) expects phishing to move there next.

A relying party that grants a session through a recovery flow and then permits every action that a passkey session permits has treated two assurance levels as equivalent, which is the defect. The remedy is not to refuse the weaker session, since recovery must remain usable, but to bound what it authorizes: ordinary operation of the account, and nothing that alters how the account is secured. Enrolling or removing a credential, changing the recovery address and moving funds should each demand re-authentication. Where they don't, an attacker holding a phished OTP acquires the same authority as the legitimate holder of the key.

Two in-the-wild kits do exactly this:

* Push Security ([Jennings, July 2025](#)) describes an AiTM kit that rewrites the identity provider's method-selection page from "passkey OR backup code" down to just the backup code wherever a weaker method coexists, naming Windows Hello for Business, Okta FastPass and Google Workspace, with Evilginx phishlets and Tycoon JS built for exactly that.

* IOActive ([Gomez, February 2026](#)) does it to Entra a second way, through a transparent Cloudflare Workers proxy tampering with the `GetCredentiaLType` response to demote the FIDO2 method or hide its row entirely.

Neither kit works against an account that has no weaker method to fall back to, which is the Mixed-Mode Problem arriving from the other direction: strict FIDO enforcement kills both outright.

Three Ways Around the Ceremony

Three attacks reach the account without defeating anything, and they're worth separating because each targets a different stage.

1. The **downgrade pretext** targets authentication: the attacker triggers a login so the victim sees a prompt, then calls as IT support about a passkey glitch and directs them to an emergency recovery portal, where a password or an SMS code is enough.
2. **Fake enrollment** targets registration: a phished QR code, presented as identity verification for some internal portal, starts a *registration* ceremony rather than an authentication one, so the victim's approval binds the attacker's passkey to the victim's account. Many identity providers permit this on username and password alone, and their "Add device" prompts are vague enough that the request looks routine. Once enrolled, the credential is indistinguishable from a legitimate one.
3. **OAuth device-code phishing** targets authorization without requiring physical proximity. The victim signs in on the genuine Microsoft page and may complete a passkey ceremony with origin binding intact, while the separate device-authorization grant delivers tokens to the attacker's requesting client. Controls must address the grant and requesting application, including Conditional Access where supported. Binding tokens to a client-held key doesn't prevent this flow when the attacker controls that client and key. The public toolkit is [SquarePhish 2.0](#), and the technique's prevalence has grown accordingly, with Push Security's telemetry recording a 37.5x increase in observed device-code phishing pages relative to its pre-2026 baseline as of April 2026, a rise the Cloud Security Alliance attributes to the commercial availability of EvilTokens, a phishing-as-a-service platform implementing the flow ([Cloud Security Alliance Labs, April 2026](#)).

User Verification and Biometric Enrollment

Relying parties validate the UV bit server-side, correctly, without establishing what a set bit actually asserts. It asserts that some enrolled biometric was presented, and never which one. [Daffalla et al.](#) state the limitation directly: "By design, FIDO2 does not support binding passkey operations to specific user biometrics. This means that the user biometrically authenticating with a passkey is not verified as the user that registered it." Any fingerprint enrolled on a device therefore satisfies user verification for every passkey on it, permanently, and the researchers registered an adversary fingerprint successfully on every service tested.

Two properties make this durable:

1. Enrollment is permissive and silent, since macOS admits three fingerprints per account, Android five per device and Windows six as observed, with no operating system notifying the owner of an addition and Windows 10 exposing no record of what is registered.
2. Removal is then constrained by identification rather than capability: macOS and Android will delete an individual entry, but label them "Finger 1" and "Finger 2" with nothing tying either to a person, which is why the paper's remedy is to wipe every biometric and re-enroll. Windows supports no per-entry deletion at all.

A related finding returns the problem to the relying party. [Daffalla et al.](#) observed that authenticating to Roblox with a synchronized credential on the adversary's device required no user verification, though the victim's original device had demanded it. Their explanation is hedged and we retain the hedge: Roblox may set `userVerification` to `discouraged`, which the paper does not confirm and which may have changed since 2024.

Interpersonal Threat Model

That gap stops being academic the moment the adversary is somebody who already holds the device and knows the PIN, which is the threat model [Daffalla, Bhattacharya, Ristenpart and Dell](#) built their paper around (USENIX Security 2025). It's the first abusability analysis of passkeys under **interpersonal threat models**, meaning intimate-partner violence, elder abuse and trafficking, where the attacker shares your home and your network and never has to phish anybody. They investigated 19 services and four passkey providers, running the adversarial-passkey protocol by hand against 12 of them, and the findings that follow are theirs.

Adding a Passkey Nobody Asked For

Under that model the cheapest move needs no phish at all, only a logged-in session, and remotely adding the adversary's own passkey to the victim's account worked on 10 of the services they tested, namely Adobe, Amazon, GitHub, Google, eBay, Intuit, LinkedIn, Microsoft, Roblox and Samsung. On six of those, adding a passkey while already signed in required no re-authentication whatsoever, so click, touch, done, and the account now carries a second key its owner never enrolled.

Notifications and Credential Revocation

Limited notifications can allow unauthorized passkeys to remain unnoticed: most services say nothing at all when a passkey is added or revoked, no service in the study sent an in-app notification for either event, and several of them, Apple and CVS among them, have no passkey security page an owner could check even if they thought to look.

Revocation is no better, since a password reset left the adversary's passkey alive on every service tested but eBay, and of the services where revocation worked at all only Amazon killed the adversary's live session inside five minutes, which leaves the victim performing the one action they're certain will help and getting no sign that it didn't.

Sitting on top of that silence is also a **gaslighting** layer, where an adversary who spoofs a user-agent, runs a VPN to fix the apparent location and edits a passkey's label can shape what the victim sees on that page when they finally do check, with no exploit and no privilege escalation anywhere in it, which is exactly what puts it in reach of somebody with no technical background at all.

One Accidental Control

One defensive note is worth handing the blue team, because it's the rare case where a product constraint accidentally became a control. TikTok and Twitter cap an account at a single active passkey, and wherever the victim already had one enrolled that cap blocked the *addition* outright, which is a narrower win than it sounds, since a copy of the credential that's already there walks straight past it. Within those limits a hard cap on active credentials still turns a silent addition into a visible conflict.

Alex and Billy

The paper closes on a scenario worth repeating. Alex and Billy are partners who share a phone PIN, which is ordinary and which no security team gets to legislate away. It goes like this:

1. Alex goes to take a shower.
2. In that window, Billy AirDrops a copy of Alex's TikTok passkey to his own iPhone, an export Apple supports and one that stays alive on Billy's phone no matter what happens to the original.
3. Billy starts reading every message.
4. Alex gets suspicious, resets the password, deletes the passkey and enrolls a fresh one.
5. None of it helps.

Three things defeat her, all of them passkey-specific:

1. **An account can hold several active passkeys at once.** Removing hers leaves his sitting there.
2. **The adversary signs in from the same household and the same network.** Every behavioral signal reads as normal.
3. **Passkeys are the strong method, so the platform often raises no alert at all.** The researchers saw no visible change on the security interfaces of Amazon, eBay, TikTok or Twitter when a cloned-passkey login happened.

The authors say it best as in the worst cases, they write, a victim has "no way to restore security of their account."

What the Person Holding the Passkey Is Left With

Daffalla et al.'s findings make device access, biometric enrollment and credential revocation part of the assessment, alongside protocol and implementation security.

Test-Case Reference

The following test cases provide a starting point for assessing a passkey deployment. Their applicability depends on the architecture, the relying party's stated policy, and the agreed scope. Completing this table doesn't establish that the deployment is secure.

Safety and interpretation

Use authorized test accounts and controlled credentials, including both identities in cross-account tests. Destructive tests require explicit approval, disposable accounts or authenticators, and an agreed recovery procedure. Physical relay deployments, social engineering, and cloud-account takeover attempts require separately defined scope.

Establish a successful control before each negative test. When testing a field covered by a signature, produce a correctly signed response with the altered value unless signature corruption is the test itself. Rejection of an invalid signature doesn't demonstrate that the server checked the altered field.

Cleanup

Record the credentials, enrollments, grants, and settings created or changed during testing. Remove test registrations at the RP and their copies from authenticators or credential providers, including credentials created locally before the RP rejected registration. Restore changed settings and verify that legitimate access still works.

Terminate test sessions and revoke issued grants or tokens through the provider's supported mechanisms, checking for any remaining access. Remove test extensions and relay infrastructure. Deleting a local token or resetting a password alone doesn't establish that access has been revoked.

Flags

Flags are cumulative and describe the most consequential subtest in a row. A prerequisite applies to the subtest that needs it; an observational first step doesn't inherit every requirement of the active steps that follow. Shared test setup and its cleanup apply throughout.

- `read-only`: inspects existing traffic or configuration, or performs nondestructive information queries. It doesn't imply an absence of logs, routine counter updates, alerts, or rate limits.
- `tamper`: modifies, forges, relays, or replays protocol messages. It makes no claim about lasting effects and can appear alongside `leaves-state` or `destructive`.
- `leaves-state`: can create or change credentials, enrollments, grants, or configuration requiring explicit cleanup, including on the tester's own accounts and devices.
- `destructive`: can remove credentials, prevent authentication, or make data inaccessible. Recovery may require administrative intervention or re-enrollment and may not be possible.
- `needs-xss`, `needs-extension`, `needs-session`, `needs-admin`, and `needs-hardware`: identify required capabilities. A session requirement may refer to the controlled victim browser, rather than a session the attacker possesses.
- `needs-proximity`: requires equipment within the relevant radio range of the test authenticator. Forwarding traffic to a remote server doesn't remove the requirement for a nearby receiver.
- `red-team`: identifies an operational exercise requiring scope beyond ordinary application testing, such as phishing or deployed relay infrastructure.
- `lockout`, `persistence`, and `privacy`: identify specific potential outcomes.

#	Actor	Test Case	Flags	What You're Looking For
1	Authenticator	AAGUID inventory	read-only	Inspect the AAGUID at bytes 37-52 of authData in an existing registration capture. Record the reported authenticator model and the RP's documented restrictions. Treat an AAGUID as an unverified model claim unless trusted attestation substantiates it under the RP's policy.
2	Authenticator	Software-authenticator acceptance	leaves-state	Register a credential using a software authenticator, such as Chrome DevTools' virtual authenticator. Does acceptance violate a documented hardware-backed or authenticator-model restriction? Without such a policy, software-authenticator support isn't itself a finding. Remove the test credential from the RP and the virtual authenticator.
3	Authenticator	CTAP management surface	destructive needs-hardware lockout privacy	On a disposable roaming key, distinguish unauthenticated reset/probing from credential-management enumeration and deletion, which normally require PIN/UV authorization. For the CTRAPS API-confusion variant, interpose between an authorized client and the authenticator and check whether the user's authorization can be reused for an unintended operation. A factory reset can invalidate credentials for every RP using that key.
4	Authenticator	Lost-key exposure (credProtect)	read-only needs-hardware privacy	Using an authorized test key, send <code>authenticatorGetAssertion</code> without PIN/UV authorization for a supplied list of candidate RP IDs. Record any disclosed credential IDs or user IDs. The query tests one RP ID at a time, while full credential-management enumeration is covered in row 3. Check the credential's actual <code>credProtect</code> level, since Chromium requests 0x02 for discoverable credentials but support and registration behavior vary.

#	Actor	Test Case	Flags	What You're Looking For
5	Authenticator / protocols (fired at the RP)	Algorithm negotiation and downgrade	tamper leaves-state	(a) Inspect pubKeyCredParams for deprecated, unassigned, or inappropriate COSE algorithms, including RSASSA-PKCS1-v1_5 with SHA-1 and symmetric HMAC. Distinguish configuration hygiene from a demonstrated downgrade. (b) Attempt a validly formed registration using an unadvertised algorithm and check that the returned algorithm is compatible with the credential key and the RP's offer. (c) At authentication, verify that signature validation uses the algorithm stored for the credential, rather than a client-supplied choice or an alternative selected after verification fails. Remove credentials created during registration tests.
6	Hybrid transport	Is hybrid even needed here?	read-only	Review whether supported clients offer hybrid sign-in and whether the deployment needs it. For any claimed restriction, identify what enforces it. A hidden UI option or transport hint doesn't establish a server-enforced restriction. Availability alone doesn't demonstrate a vulnerability.
7	Hybrid transport	QR relay feasibility	tamper needs-proximity red-team	Display a fresh FIDO : / QR from a legitimate test login on a controlled page and have the test user scan and approve it. The attacker's client or relay receiver must be within BLE range of the authenticator. Check whether the attacker's session becomes authenticated, since forwarding the QR alone isn't sufficient.
8	Hybrid transport	FIDO : / deep-link handling	tamper needs-proximity	Test whether untrusted web content or a WebView can dispatch a FIDO : / URI into a passkey prompt, accounting for the platform's intended app-link behavior. CVE-2024-9956 is the patched browser example. For an end-to-end relay, satisfy row 7's BLE-proximity requirement and verify authentication in the attacker's session, since opening the prompt alone doesn't demonstrate account access.
9	Hybrid transport	Transport restrictions are a hint	tamper	Modify or ignore transports in allowCredentials and test any claimed restriction with a client under your control. The field is an unsigned client hint, so it cannot establish how the ceremony was transported. Ask what independently enforces the restriction or supports a detection claim. Use row 7 for an end-to-end relay test.

#	Actor	Test Case	Flags	What You're Looking For
10	Client	Extension and API hooking	tamper leaves-state needs-extension	Use a test extension with Chrome's <code>webAuthenticationProxy</code> permission to intercept <code>navigator.credentials.create()</code> and <code>.get()</code> in a controlled browser profile. Determine whether extension-generated responses bypass the RP's registration or authentication controls. The registration branch can create credentials. Detach and unload the extension, and remove any resulting test credentials afterward.
11	Client	Framing and clickjacking (crossOrigin, topOrigin)	tamper	Using a previously enrolled test key, submit validly signed authentication assertions with <code>crossOrigin: true</code> and an unapproved <code>topOrigin</code> . Does the RP reject unexpected cross-origin embedding? For deployments that intentionally support cross-origin frames, include an approved top origin as a positive control.
12	Client	Challenge-to-session binding under XSS	tamper needs-xss	With controlled script execution on the RP origin, obtain an assertion for a challenge issued in test session A and submit that assertion from test session B. Determine whether the RP binds the challenge to the initiating session and whether cross-session acceptance permits unauthorized access. Limit the conclusion to cross-session substitution, since session binding doesn't eliminate arbitrary XSS abuse.
13	Client	Post-auth session security	tamper needs-xss needs-session	In an authenticated test session, check whether injected JavaScript can read the session cookie and whether an extracted cookie works from a second client. If <code>HttpOnly</code> prevents extraction, use a non-mutating authenticated request to test session riding. Evaluate any claimed session-binding control, recognizing that device fingerprints alone don't provide cryptographic binding. Sensitive-action authorization is covered in row 41.
14	Client	Mobile app and WebView misbinding	tamper	Identify whether the flow uses browser-origin checks or a native-app association mechanism, then test the relevant RP-ID and origin restrictions with a controlled client. Record an accepted unauthorized origin or app association rather than inferring failure from the absence of a URL bar.

#	Actor	Test Case	Flags	What You're Looking For
15	Client	Passkey as the vault opener	destructive leaves-state needs-session lockout	On a disposable vault, test authorization for enrolling or replacing the passkey that unlocks the manager itself, including enrollment from a second device. Determine how the credential gates release or derivation of the vault's encryption key. Evaluate this against the supported unlock design rather than treating a missing master-password prompt as sufficient evidence of a bypass. Remove test enrollments and verify that the original unlock method and vault decryption still work.
16	Relying party	Can you read the traffic at all?	read-only	Using an existing capture, decode and re-encode the ceremony offline, including the vendor envelope and every Base64 layer. Confirm that the unmodified data round-trips byte for byte before relying on negative test results. No registration or authentication request needs to be resubmitted for this check.
17	Relying party	Signature verification	tamper	For each variant, obtain a fresh authentication challenge and prepare an otherwise valid response. Corrupt the signature, omit it, or substitute a signature generated by a different test credential. Confirm rejection without account access, and compare with a separate valid control ceremony.
18	Relying party	Origin and RP-ID hash validation	tamper	For each variant, use a fresh challenge and a controlled test key to sign an assertion with a changed <code>clientDataJSON.origin</code> . Vary scheme, host, and port separately, including unauthorized subdomains, look-alike hosts, and non-default ports. Verify enforcement of the complete browser-origin allowlist. Separately change <code>rpIdHash</code> while retaining an allowed origin and re-sign the assertion to test RP-ID hash validation independently.
19	Relying party	rpId scope	read-only	Record the effective RP ID and identify the subdomains from which browsers can request credentials scoped to it. Where the scope extends beyond the intended application, inventory less-trusted or abandoned subdomains and assess the exposure alongside the origin allowlist in row 18. Claiming a domain requires separately approved active testing.

#	Actor	Test Case	Flags	What You're Looking For
20	Relying party	Related Origin Requests allowlist	read-only	Retrieve the /.well-known/webauthn document for the RP ID and inspect its related-origin list. Identify expired, unowned, or registrable domains and record which relying-party relationships they affect. Keep domain acquisition and active takeover validation outside this read-only check.
21	Relying party	UV and UP enforcement	tamper	(a) Keep server-side requirements unchanged. For a correctly signed UV=0 assertion, change the client-side userVerification request to discouraged before the ceremony where the authenticator and credential permit it, or use an enrolled controlled authenticator. Use controlled tooling for correctly signed UP=0 assertions where presence is required. The audit found 10 RPs accepting UV=0 despite requiring verification, and 27 accepting UP=0. (b) Corrupt only the signature of an otherwise policy-compliant assertion and confirm rejection. A post-signing flag flip also invalidates the signature and cannot isolate UV/UP enforcement. Reading the flags before signature verification isn't itself a vulnerability if a failed signature still prevents acceptance.
22	Relying party	Challenge freshness and session binding	tamper	Test reuse by replaying a previously accepted assertion. Test expiration separately by holding an unused, valid response until its challenge exceeds the RP's configured lifetime. For session binding, submit a valid assertion issued for session A from session B. Record whether each case grants account access, rather than interpreting an HTTP success response as proof of authentication.
23	Relying party	create versus get confusion	tamper leaves-state	Submit otherwise well-formed registration and authentication responses with the wrong clientDataJSON.type for the endpoint. Preserve valid signatures wherever the selected response format requires them, so rejection isn't explained by a schema or signature error. Confirm that the RP distinguishes webauthn.create from webauthn.get. Registration-side probes can create credentials and require cleanup.

#	Actor	Test Case	Flags	What You're Looking For
24	Relying party	Attestation and AAGUID enforcement	tamper leaves-state	Under a policy requiring verified authenticator provenance, submit <code>fmt : none</code> , packed self-attestation without <code>x5c</code> , and certificate-backed attestations with invalid signatures or untrusted chains, including claims of an allowlisted AAGUID. Confirm rejection of evidence that fails the RP's trust and model policy, using a valid policy-compliant registration as a positive control. Where available, also test a validly attested but disallowed model. Evaluate the returned evidence regardless of conveyance preference, including <code>indirect</code> ; requesting <code>direct</code> or <code>enterprise</code> isn't enforcement. Remove accepted test registrations and any local credentials created during rejected attempts.
25	Relying party	Credential-ID uniqueness (Credential Overwrite)	destructive leaves-state lockout	Using two disposable accounts, attempt registration with a credential ID already associated with the other account. Test ID reuse alone and, where applicable, reuse of the ID and public key together. Determine whether the RP rejects the collision or overwrites, appends, deletes, or misbinds credential records. Verify both accounts after each attempt, allowing for administrator-assisted recovery or recreation of the test accounts.
26	Relying party	Registration session binding	leaves-state persistence	For an existing test account, attempt registration without its authorized session and from a second account's session. Substitute account identifiers where the API accepts them, then check whether a successful response actually binds a usable credential to the wrong account. Verify challenge-to-account and challenge-to-session binding, and remove any resulting test credentials.
27	Relying party	Registration CSRF	leaves-state needs-session persistence	With the target test account signed in, use a real cross-origin browser request to attempt enrollment of an attacker-controlled test credential. Check cookie behavior, CSRF defenses, and challenge binding, then verify whether a usable credential was added to that account. A request replayed only through a proxy doesn't establish browser-based CSRF exploitability. Remove any resulting test registration.

#	Actor	Test Case	Flags	What You're Looking For
28	Relying party	Enrollment step-up	leaves-state needs-session persistence	For an account with an existing strong factor, test whether another passkey can be added from an active session without step-up. Compare sessions established with that factor against sessions obtained through recovery or weaker fallback methods. Check whether the UI distinguishes enrollment from sign-in, and record any path from lower-assurance access to persistent credentials. Remove the test enrollment.
29	Relying party	Admin-API and IdP enrollment	leaves-state needs-admin persistence	Inventory principals that can provision passkeys through the IdP's administrative API and assess the users they can target. For Entra ID's provisioning API , review both <code>UserAuthMethod-Passkey.ReadWrite.All</code> and the broader <code>UserAuthenticationMethod.ReadWrite.All</code> , distinguishing application grants from delegated access and its role requirements. In a controlled test, verify whether that authority can enroll a credential for a higher-privileged account. Revoke the test authentication method and terminate resulting sessions.
30	Relying party	Credential-management authorization	destructive leaves-state needs-session lockout	Using two test accounts, exercise credential-listing, renaming, and deletion endpoints with identifiers belonging to the other account. Verify authorization separately for each operation. Listing can expose metadata, renaming leaves durable changes, and deletion can prevent access. Restore labels and replace deleted test credentials as needed, checking that neither account remains impaired.
31	Relying party	signCount enforcement	destructive leaves-state lockout	Using a disposable credential, obtain its private key or control an authenticator capable of signing chosen counter values for that credential. Submit correctly signed assertions with non-increasing counters and, separately, an inflated counter. Record the RP's observable response and any available evidence of counter updates or risk handling. A fast-forward lockout requires the RP to store the inflated value and reject subsequent legitimate assertions because their counters don't exceed it. Zero-only counters and acceptance of non-increasing counters aren't automatically defects. Retire and replace an affected test credential rather than assuming its previous counter state can be restored.

#	Actor	Test Case	Flags	What You're Looking For
32	Relying party	Spec housekeeping	tamper leaves-state destructive	(a) Substitute another test account's <code>userHandle</code> and verify credential-ownership checks. (b) Test empty and oversized credential IDs against the 1,023-byte maximum and any additional bounds imposed by the deployed specification or profile. (c) Submit <code>fmt: none</code> with a nonempty <code>attStmt</code> and confirm rejection. (d) Evaluate Token Binding handling against the deployed version and policy. Level 3 reserves the field without requiring blanket rejection. Use bounded inputs and disposable accounts, because malformed registrations can leave records that require administrative removal.
33	Relying party	Library fingerprint and leftover endpoints	read-only	Identify the exact WebAuthn library build and compare it with the advisories cited in this paper, including applicable vendor backports. Check both Keycloak attestation issues, CVE-2025-12150 and CVE-2026-6856, rather than relying on a single version cutoff. Inventory default registration and login endpoints left enabled alongside custom routes, without submitting registration or authentication responses. Reproduce any suspected bypass under the corresponding active test.
34	Sync fabric (fired at the RP)	BE/BS consistency	tamper leaves-state	Register a disposable credential with <code>BE=0</code> , then submit a correctly signed assertion claiming <code>BE=1</code> . Where backup state affects RP policy, verify consistency with the stored registration value. Separately test <code>BE=0</code> , <code>BS=1</code> , which is invalid. For a separate <code>BE=1</code> registration, use <code>BS=0</code> and <code>BS=1</code> as protocol-valid controls, accounting for any stricter RP policy. Remove the test registrations afterward.
35	Sync fabric	Synced where policy says bound	leaves-state	Where policy requires device-bound credentials, attempt registration with a known synced provider and check whether the RP rejects it or permits a documented exception. Identify the evidence used to enforce the restriction, including any required attestation, rather than treating an unverified authenticator claim as proof. Remove any test credentials created at the RP or provider.

#	Actor	Test Case	Flags	What You're Looking For
36	Sync fabric	Provider and vault posture	read-only	Record how the credential is protected at rest, during synchronization, and when used for signing. Identify which devices and shared-vault members can access it, and whether export or migration through mechanisms such as the Credential Exchange Format is supported. Base hardware-protection conclusions on the specific implementation. Device enrollment, export, and configuration changes require separate active testing.
37	Sync fabric	Cloud-account takeover path	read-only	Review the authentication, recovery, and device-enrollment controls protecting the provider account, including any separate check that releases encrypted passkey material. Determine whether account recovery or a phished provider credential could satisfy those checks. Keep actual account takeover, PIN guessing, and new-device enrollment outside this read-only review and subject to separately agreed scope and cleanup.
38	User and recovery	Downgrade path	tamper needs-extension	Exercise the fallback path when a passkey is unavailable and record which alternative methods are offered. Separately use a test extension to block the WebAuthn API and observe whether the application fails closed or enables a weaker method. Evaluate the resulting session against the deployment's assurance policy, then remove the extension and terminate test sessions.
39	User and recovery	Method-selection rewrite	tamper red-team	For an authorized phishing simulation, suppress the passkey option in the method-selection response or page while retaining an available weaker method. Verify whether the IdP issues an authenticated session through that method and whether it satisfies the intended access policy. Distinguish a permitted fallback from a bypass of enforced phishing-resistant authentication. Terminate resulting test sessions.

#	Actor	Test Case	Flags	What You're Looking For
40	User and recovery	Recovery as de-facto authentication, and eviction	destructive leaves-state persistence lockout	Using controlled recovery channels and disposable accounts, test whether recovery can remove existing credentials or enroll replacements. Check password-reset effects, session invalidation after passkey revocation, and account-holder notifications separately. In Daffalla et al.'s 2025 study , only eBay revoked passkeys on password reset, and only Amazon ended tested sessions within five minutes of passkey revocation. These results describe that sample, so verify current behavior and restore the test account afterward.
41	User and recovery	Session assurance level	tamper leaves-state needs-session	Select an operation with a defined step-up requirement and compare sessions established through passkeys, recovery, and other supported methods. Verify that the server enforces the required assurance at the time of the action, using the actual factors and session context rather than method labels alone. Use a reversible test operation on a disposable account, then restore changed state.
42	User and recovery	Alternate grants	tamper leaves-state red-team persistence	Review alternate authorization grants, particularly OAuth device authorization, that can deliver tokens to an attacker-controlled client after the user completes a legitimate sign-in. In an authorized simulation, verify which client receives tokens, what scopes and refresh capability they carry, and which grant or consent records persist. Assess grant restrictions and approval controls separately from WebAuthn origin binding. Revoke test grants and sessions, and verify when issued tokens cease to work.
43	User and recovery	Helpdesk and support channels	destructive leaves-state red-team lockout	Separate review of support procedures from a live reset attempt. In an explicitly authorized exercise using a designated test identity, determine whether publicly available information is sufficient to obtain a passkey reset or a recovery-setting change. Stop at the agreed boundary, record notifications and any access issued, and restore affected test credentials and recovery settings.

The Tooling

Tool support varies across actors. The following table summarizes the available options.

Actor	Tool	What it gives you
Authenticator	python-fido2	Yubico's CTAP2 and WebAuthn client/server library for communicating with authenticators and building custom test tooling. Rogue-authenticator emulation requires an additional implementation.
Authenticator	CTRAPS	Casagrande and Antonioli's toolkit, published with the paper. Reproduces credential enumeration, credential deletion and factory reset against a key you own.
Authenticator	ProVerif , Tamarin	Formal verification. The right tools for this layer and the wrong tools for your engagement.
Hybrid transport	CrossCheck	A research artifact rather than a maintained tool. Drives a headless browser at the real relying party, decodes the FIDO: / URI out of the QR and re-encodes it, with the non-discoverable-credential flag as the only knob. No caBLE, no Bluetooth, so the BLE leg is yours to build.
Client	Chrome DevTools virtual authenticator	Supports controlled registration and authentication tests. <code>setUserVerified</code> controls whether a UV check succeeds; <code>setAutomaticPresenceSimulation</code> controls whether presence checks complete automatically or remain pending. These settings don't by themselves guarantee a validly signed UV=0 or UP=0 response. For RP enforcement tests, use a client/authenticator setup that produces the intended flags and verify the response signature before submission.
Relying party	webauthn-cbor	Decodes, and stops there.
Relying party	PasskeyScanner	Passive detection plus a visualization of <code>authData</code> , so you're not reading hex by hand.
Relying party	Passkey-Raider	Decode-and-tamper by hand, and it can plant a key and re-sign.
Relying party	Burp_FIDO2	Out of Chen's MSc thesis. The broadest attack matrix of the Burp Suite extensions, and it struggles the moment a target uses a custom encoding.
Relying party	PASSKEYS-ATTACKER	The Bochum audit's own harness (Apache-2.0, Zenodo DOI 10.5281/zenodo.17898769). Emulates client and authenticator, mutates every field, and ships RP discovery plus a vulnerable practice platform. Hosted companion at passkeys.tools .
Sync fabric, user and recovery	VaultJacking, Evilginx phishlets, the IOActive Workers proxy, SquarePhish 2.0	Campaign infrastructure rather than pentest tooling. Worth naming in the report either way, so the client knows which half of their risk went untested.

Actor	Tool	What it gives you
Enterprise and Windows	Pass-the-Passkey	Grafnetter's Black Hat USA 2026 research, out since July 2026 with slides and a whitepaper . The paper documents a historical Windows 11/Entra ID replay chain. In affected Windows builds, logged assertions were readable by local Users-group members; remote retrieval required additional group membership. Replaying those assertions against the tested Entra ID configuration could impersonate a privileged cloud user. Windows patched the logging issue on July 14, 2026 by truncating logged signatures. Entra's May 2026 mitigation added counter tracking for compatible credentials, while the paper reported residual replay for zero-counter credentials. The rest of the family covers assertion injection, browser hooking, prompt flooding and a Credential UI window-handle spoof still unfixed at publication. Tooling at pass-the-passkey and webauthn-interop .

Ours is [Passkey Editor](#), a Burp Suite extension built around the encoding wall: it profiles the ceremony per host, unwraps the chain while remembering each layer's exact flavor and padding, decodes the CBOR and COSE into an editable tab, and re-signs on the way back out. No single move there is a first, and having all of them in one workflow is what the others are missing. The full account is in [Introducing Passkey Editor](#).

Conclusion

Passkeys provide a substantial improvement over password-based authentication, reducing exposure to credential stuffing, password spraying and conventional credential phishing. Their deployment still requires the surrounding controls examined in this paper.

An assessment should cover credential enrollment, client behavior, relying-party verification, synchronization and recovery alongside the ceremony itself. The six-actor model and test-case reference support that assessment, with outcomes interpreted against the deployment's threat model and policy.

References

The series

- Giordano, Demystifying Passkeys, Under the Hood: The Protocol, Anvil Secure
- Giordano, Demystifying Passkeys, Under the Hood: The Architecture, Anvil Secure

Specifications and framing

- W3C Web Authentication (WebAuthn) Level 3 Specification (Candidate Recommendation Snapshot, May 26, 2026)
- FIDO Alliance, Client to Authenticator Protocol (CTAP) 2.3, February 26, 2026
- Nishant Kaushik (CTO, FIDO Alliance), Passkeys Are Not Broken. The Conversation About Them Often Is, September 2, 2025
- FIDO Alliance, The State of Passkeys 2026: Global Consumer and Workforce Report, May 7, 2026

The at-scale audit (cited throughout)

- Jannett, Mayer, Westers, Mladenov, Mainka, Schwenk, The State of Passkeys: Studying the Adoption and Security of Passkeys on the Web, USENIX Security 2026 (artifacts, Zenodo DOI 10.5281/zenodo.17898769; hosted tooling at passkeys.tools)

Actor 1: Authenticator and protocols

- Barbosa, Boldyreva, Chen, Warinschi, Provable Security Analysis of FIDO2, CRYPTO 2021
- Bindel, Cremers, Zhao, FIDO2, CTAP 2.1, and WebAuthn 2: Provable Security and Post-Quantum Instantiation, IEEE S&P 2023
- Barbosa et al., Privacy and Security of FIDO2 Revisited, PoPETs 2025
- Guan et al., A Formal Analysis of the FIDO2 Protocols, ESORICS 2022
- NIST, FIPS 204: Module-Lattice-Based Digital Signature Standard (ML-DSA), August 2024
- Bindel, Gama, Guasch, Ronen, To Attest or Not to Attest: Provable Attestation in FIDO2, ASIACRYPT 2023
- Casagrande, Antonioli, CTRAPS: CTAP Client Impersonation and API Confusion on FIDO2, arXiv:2412.02349 (EuroS&P 2025) (toolkit)
- Roche, Lomne, Mutschler, Imbert, A Side Journey to Titan (EM side-channel key extraction, CVE-2021-3011), USENIX Security 2021 / NinjaLab writeup
- NinjaLab, EUCLEAK (ePrint 2024/1380)
- Yadav, Seamons, A Security and Usability Analysis of Local Attacks Against FIDO2, NDSS 2024
- passkeys.dev, Known Issues: User Verification
- Chromium, credProtect levels requested by the browser
- Yubico, python-fido2
- Anvil Secure, Glitching STM32 Read-Out Protection With Voltage Fault Injection

Actor 2: Hybrid transport and cross-device authentication

- Kim, Shin, Ryu, Choi, HiPass: Hijacking CTAP in Passkey Authentication, IEEE Access vol. 13, 2025
- inovex, Phishing for Passkeys: An Analysis of WebAuthn and CTAP (2024)
- Kniep, FIDO Cross-Device Authentication Phishing (2025)
- Righi, CVE-2024-9956: Passkey Account Takeover in All Mobile Browsers (CrossCheck, NVD)
- Expel, An important update (and apology) on our PoisonSeed blog, July 2025

- Kniep, Bypassing the CVE-2024-9956 fix on Apple platforms (2025)
- Adam Langley, A Tour of WebAuthn (2024)

Actor 3: Client (browser, extension, password manager)

- SquareX, Passkeys Pwned: Turning WebAuthn Against Itself, DEF CON 33, 2025 / coverage
- Marek Toth, DOM-Based Extension Clickjacking (2025)
- Catalano, Chezzi, Barletta, Tommasi, Defeating FIDO2/CTAP2/WebAuthn using browser in the middle and reflected cross site scripting, J. Computer Virology and Hacking Techniques 21 (2025)
- Helme, Passkeys, Permissions Policy and Bug Hunting in 1Password's WebAuthn Wrapper (2026)

Actor 4: Relying party

Origin, scope, and cross-RP

- Squarcina, Tempesta, Veronese, Calzavara, Maffei, Can I Take Your Subdomain? Exploring Same-Site Attacks in the Modern Web, USENIX Security 2021
- Balliauw (Duende), Deep Dive: Relying Party ID and Origin With Passkeys (2025)
- Scott Helme, Open-Sourcing passkeys-php: A Security-Focused WebAuthn Library for PHP (2026)
- CVE-2026-30964, PHP web-auth/webauthn-lib origin collapsed to host-only (GHSA-f7pm-6hr8-7ggm)

Verification, flags, and implementation

- Chen, Vulnerability Testing for WebAuthn (MSc thesis, University of Twente, with Computest, March 2024) (full text)
- RUB-NDS, passkeys.tools: Encode, Decode, Intercept, Modify, and Exploit WebAuthn Operations
- U-Zyn Chua, Passkey has a theft detection feature, but Apple, Google and Microsoft broke it (2025)
- CVE-2025-26788, StrongKey FIDO Server Passkey Authentication Bypass (Securing, 2025)
- CVE-2026-46419, YSA-2026-02: Yubico java-webauthn-server cross-user success (2026) (Yubico scores it 7.7, NVD and GHSA 7.5)
- CVE-2025-12150, Keycloak WebAuthn Attestation Statement Verification Bypass (GHSA-7g5x-9c4v-4w5r)
- CVE-2026-6856, Keycloak Acceptable-AAGUID policy bypass (Keycloak issue #48388) (fixed in 26.4.12, 26.6.2 and 26.7.0)
- CVE-2024-12225, Quarkus quarkus-security-webauthn Authentication Bypass (Red Hat Bugzilla) (CWE-288, leftover default endpoints, not an origin-validation bug)
- Grafnetter (SpecterOps), Pass-the-Passkey Family of Attacks, Black Hat USA 2026 (slides, whitepaper, tooling)

Enrollment and persistence

- Microsoft Graph, Create fido2AuthenticationMethod: provisioning flow and permissions
- Rozendaal (Bureau Veritas), Abusing FIDO2 Passkeys: Entra ID Global Administrator takeover via admin-API passkey enrollment
- SpecterOps, I'd TAP That Pass: abusing UserAuthenticationMethod.ReadWrite.All in Microsoft Entra ID (2024)
- ITInnovationStation, The Passkey Pwned Attack: Breaking the "Unbreakable" (2025)

Actor 5: Sync fabric

- Blessing, Hugenhroth, Anderson, Beresford, SoK: Web Authentication and Recovery in the Age of End-to-End Encryption, PoPETs 2025(3)
- Scarlata, Torrisi, Backendal, Paterson, Zero Knowledge (About) Encryption, USENIX Security 2026 (IACR ePrint 2026/058) (project site)
- Brazzell (PhishU), Vaultjacking: One Captured PIN, the Entire Google Password Manager Vault (2026)
- Prove, Passkey Syncing Fraud: The New Attack Vector (2025)
- Spensky, Your Passkey is Weak: Phishing the Unphishable, DEF CON 33, 2025
- Islam, Arora, Chatterjee, Wang, CASPER: Detecting Compromise of Passkey Storage on the Cloud, USENIX Security 2025
- Bicakci, Varli, Korkmaz, Uzunay, QES-Backed Virtual FIDO2 Authenticators, arXiv:2601.06554 (2026)
- Bonneau, Herley, van Oorschot, Stajano, The Quest to Replace Passwords, IEEE S&P 2012
- Büttner, Gruschka, Device-Bound vs. Synced Credentials: A Comparative Evaluation of Passkey Authentication, ICISSP 2025
- FIDO Alliance, Credential Exchange Format and Credential Exchange Protocol, Working Drafts (2024)

Actor 6: User and recovery

- UK NCSC, Comparing the security properties of traditional user credentials and FIDO2 credentials for personal use (April 2026)
- Proofpoint, Don't Phish, Let Me Down: FIDO Authentication Downgrade (2025)
- Jennings (Push Security), MFA downgrade: How attackers are getting around phishing-resistant authentication (2025)
- Gomez (IOActive), Authentication Downgrade Attacks: Deep Dive into MFA Bypass (2026)
- Spicer (Netcraft), Phishing After Passkeys: What Attacks to Expect (2026)
- Romsdahl, Talebzadeh, SquarePhish 2.0: QR Code OAuth 2.0 Device Code Flow Phishing (Disobey 2026) (tool)
- Cloud Security Alliance Labs, OAuth Device Code Phishing: 37x Surge in Enterprise ATO (April 2026)
- Daffalla et al., A Framework for Abusability Analysis: The Case of Passkeys in Interpersonal Threat Models, USENIX Security 2025
- KnowBe4, You Are Still Vulnerable to Password Attacks When Using Passkeys

Measurement, adoption, and market data

- Bhardwaj, Sastry, State of Passkey Authentication in the Wild: A Census of the Top 100K Sites, PAM 2026 (Springer LNCS 16477)

Tooling

- PasskeyScanner, Passkey-Raider, webauthn-cbor
- Burp_FIDO2
- Yubico python-fido2
- CrossCheck
- ProVerif, Tamarin
- Spensky, yourpasskeyisweak PoC
- Grafnetter, webauthn-interop / DSInternals.Passkeys
- CTRAPS toolkit

- [PASSKEYS-ATTACKER and the State of Passkeys artifacts](#) (Apache-2.0, Zenodo DOI 10.5281/zenodo.17898769), hosted at [passkeys.tools](#)
 - [Passkey Editor](#), introduced in [Giordano, Introducing Passkey Editor, Anvil Secure](#)
-

About the Author

Matteo Giordano is an Italy-based offensive security specialist and Security Engineer at Anvil Secure, focused on application penetration testing and offensive research, with a growing focus on AI security and AI red teaming. He came into security from kernel-side development at *rev.ng Labs*, where he worked on a next-generation decompiler. While researching at the intersection of AI and security, on the sidelines, for the past year he has also researched WebAuthn and passkeys from an attacker's perspective, mapping the real-world attack surface that sits around the protocol's cryptographic core.

About Anvil Secure

Anvil was founded in 2016 with a vision to make a change in the information security consulting services industry. Anvil has since grown to be an industry-recognized information security partner to some of the largest tech and Fortune 500 companies across the globe.

Anvil was founded with the principles of creating an information consulting services company that is honest, transparent, professional, and that will consistently deliver quality services to our clients. Anvil continues to hold these principles to this day and is now known for delivering consistently quality service, and for our transparent and inclusive approach.

Anvil's team is filled with dedicated industry veterans that are experts in fields such as:

- Embedded/hardware security
- Industrial control systems
- Cloud security
- Web application security
- Mobile security
- Network security
- Operating system security

Several team members come from National Labs and other industry-recognized consulting firms. The team's technical backgrounds and consulting experiences position Anvil to effectively improve the security posture of leading technology companies and security groups.

Anvil is headquartered in Seattle with an office in Amsterdam as well as several employees located around the globe including Italy, France, Spain, and Argentina.

For more information about Anvil and to stay up to date on our latest research and progress, visit our website and social media pages:

- [Website](#)
- [LinkedIn](#)
- [Twitter](#)

For any questions or further information, please reach out via email or phone:

- Email: info@anvilsecure.com
- Phone: 206-753-7649