Linux Passkeys Update

An update on improving passkey support in Linux

2025-05-19

With the announcements from big companies at World Password Day about passkeys, I thought I should share what I've been working on for passkey support on Linux. I've been putting off writing this down because I've been busy with other things, but perhaps if I write things out, other people will have a clearer idea of the work to do and can help where I can't right now. This will be less polished than I would like, but hopefully it'll be good enough for now and I can refine it later.

Table of Contents

WebAuthn Overview

On and off for the past couple of years (has it been that long?), I've been working on a FIDO2/WebAuthn platform API for passkeys. I'm assuming you're familiar with passkeys and WebAuthn, if not, you can read Adam Langley's excellent "Tour of WebAuthn" for a comprehensive summary of the WebAuthn API, or Trail of Bits' recent article as a shorter overview of its security model.

WebAuthn Roles

Roughly, besides the user, there are 4 roles in WebAuthn transactions:

- relying party (website)
- client (browser or app)
- platform (may or may not be separate from the client)
- authenticator (may or may not be separate from the platform)

As an example, on Windows, say you're trying to login to Foo Store on `https://example.com`, on Edge using a security key. In this situation, the relying party is Foo Store, identified by the relying party ID (`rpId`) `example.com`, the client is Edge, and the authenticator is your FIDO2 security key. But the platform is actually Windows Hello.

Windows Hello is a platform?

You may be familiar with Windows Hello being the thing you use to sign into your computer with a fingerprint, face ID or a PIN. It can authenticate you to your own device via these methods, but it also has two other roles, what I'll call a WebAuthn *platform API* and a *platform authenticator*. As a platform API, Windows Hello mediates WebAuthn requests from clients running on the OS (like browsers or Windows apps). This gives a uniform interface for users to use passkeys on other authenticators, like a security key or smartphone, which is great for usability. It also serves as a platform authenticator: it can store WebAuthn credentials on the Windows machine and use them to sign WebAuthn requests. This is built into the device (and maybe synced[1]), so as long as you have access to the device, you have access to your credentials, which is also good.

For the rest of this article, I will use "platform" to refer to the platform API, and will specifically use "platform authenticator" for the internal authenticator provided by the platform.

Trust boundaries between WebAuthn roles

A great security model to provide to users would be:

> An authenticator will only sign a request for a relying party if the user requests via the relying party's application that they want a signature for that relying party.

The key phrase is the part "via the relying party's application": this is the crux of the phishing resistance property of passkeys. Perhaps if the client, platform and authenticator and UI were all one packaged application, this would be easy. But since the authentication request passes between multiple roles fulfilled by different processes, we must take steps to make sure that the context of the user's request is propagated correctly.

Here is an overview of the flow the request must take:

1. The client receives challenge for an `rpID`'s credential from RP from some `origin`. Client passes challenge and `origin` to platform.
    - Browsers derive `origin` through normal origin validation over HTTPS.
    - Native apps (including apps on mobile OSes and macOS) verify the origin through app IDs attested by developer and vendor signatures.
2. The platform verifies the client is legitimately calling this on behalf of a user.
    - Android, iOS/iPadOS and macOS approximate this by instead designating whether the app is trustworthy. Under normal circumstances, an app cannot run if it is not verified by Google/Apple. If it can't run, it can't request a signature. Easy! (This glosses over privileged system services.) These platforms also

require a permission in the signed app manifest that allows requesting credentials for any origin (for browsers) or for specific origins bound to the app.
    - Windows will collect the executable's path and any signed attributes at this time, but unsigned executables may still invoke the platform API.
3. The platform verifies the `rpId` matches the `origin`.
    - For browsers on desktop OSes, the WebAuthn API allows a 1:* relationship between an `rpId` and a set of origins, using Related Origin Requests. This is verified by the platform on some OSes, and by the browser in others.
    - As mentioned above, Android and Apple OSes limit which apps are allowed to request specific origins. Browsers are given unrestricted access. Other apps bind the app's ID to the `rpId` via the app's signed manifest and a file served over HTTPS on the origin(s) corresponding to the `rpId`. This also allows a 1:* relationship between an `rpId` and valid origins.
4. The platform prompts the user to select an authenticator for `rpId`.
    - By now, the platform trusts that the `rpId` was delivered by a trusted client, and should show the user that `rpId` and relevant client information in the prompt. and the `rpId`.
    - The platform must make an effort to make this prompt unspoofable and disallow concurrent requests to prevent user confusion.
5. The user verifies that the `rpId` is expected and selects an authenticator.
    - The platform must verify that the user's response to the selection prompt came from the platform's previous prompt.
6. The platform sends challenge to the selected authenticator.
    - The platform/OS should ensure that authenticators are only accessible by the platform, not random user processes, e.g. by filtering FIDO HID usages for USB, Bluetooth service data for Bluetooth/hybrid, stored credentials for platform authenticators, etc. from user processes.
7. The authenticator verifies that the user is present, then signs the challenge.

At this point, the signature is passed back up through platform and client to the RP. The RP can verify the signature, and everybody's happy.

If we skip some of these steps, here's a few things that could go wrong:

- A web app served from a typo'd domain or an malicious app lookalike could request a credential for the authentic app, and the authenticator would just sign it for the malicious app.
- Any script could invoke the WebAuthn API and start the flow, giving the attacker an easy way to mount phishing attacks.
- A malicious process could send a request directly to an authenticator to sign, bypassing the platform API, and somehow tricks the user into asserting user presence (or just signing the data with the key directly, in the case of on-device credentials).

- An attacker could show a prompt that looks like a normal WebAuthn credential request, but sends a challenge for a different origin than displayed in the prompt.

Some features this requires:

- application identity: the OS has to be able to provide the platform with the application identity so that it can make policy decisions about which `rpId`/origin
- remote lookup of origin bindings: the platform must make calls out to external sites to match bindings.
- unspoofable UI: the user has to be able to trust the UI prompt
- exclusive access to authenticators: since policy checks are implemented in the platform, credentials can only be secure if requests are mediated by the platform.

State of Other WebAuthn Platforms

Windows, macOS/iOS and Android all have platform APIs and platform authenticators. This wasn't always the case, though. The OS vendors took a little while to implement native platform support, so in order to bootstrap the passkey ecosystem, browsers took on both the client and the platform [API] roles. Chrome and Firefox used OS APIs to directly connect to security keys over USB, and later Chrome added support for Bluetooth and "hybrid"/smartphone devices and even internal passkeys secured by biometrics. As the OS vendors added more features, the browsers deferred more responsibility for handling WebAuthn requests to the OS.

This is a subset of features[2] that are provided in the Windows, macOS, iOS/iPadOS, and Android platforms:

- Save passkeys on the device platform authenticator (perhaps syncing to other devices)
- Handle "cross-platform" authenticators over different transports (USB, BLE, NFC?, and hybrid/caBLE)
- Third-party passkey providers (Windows in beta)
- App-Origin binding (not on Windows)
- Related origin requests (ChromeOS and iOS/iPadOS only?)

This is great! Users get a unified experience on their device across browsers and apps[3], perhaps with backups enabled, and browsers benefit from not having to do that work themselves.

State of Linux WebAuthn Platform

Current State

However, in Linux-land, things aren't so great. Chrome still has the internal WebAuthn request handling code built-in, so Chrome/Chromium users on Linux can still use many different types of authenticators, but Firefox has only implemented USB so far. So Linux users may have to decide between using their preferred browser, or having access to their accounts, which is not a great position. (Password managers that support passkeys using browser extensions give more user choice, but we'll get to that in a minute.)

Besides that, many Linux distros are leaning into sandboxed applications, whether using Flatpaks, Snaps or other sandboxing technology. But in order for browsers to provide FIDO2 authenticator support, they have to have raw access to devices, which kinda blows a huge hole in the sandbox. It would be great if we could close that hole.

There is not a platform authenticator to speak of on Linux, so users cannot create device-bound passkeys, even though the hardware (TPM) to do so is common in modern devices.

As for syncing passkeys, users must use Google Password Manager built into Chrome, or a password manager extension. I think this is satisfactory for user choice, but not for user experience. When using a password manager extension, the website, browser and extension all kind of "fight" for the UI, which can be confusing. More than that, extensions require a lot of privilege in order to inject their scripts for filling passkeys/passwords into the web page, which can introduce vulnerabilities. While some password manager vendors handle this privilege with care, not all of them are equal. Having these APIs built into the browser can remove some of the security concerns around browser extensions.

What We're Doing About It

When I began working on this, I was inspired by previous work by Daiko Ueno and Norbert Pocs and Alfie Fresta. Alfie had started building a library to interact with authenticators over different transports, but there was not yet a common API for browsers and apps to call the library. On Linux desktops, "portals" are the common way to provide an API with fine-grained access. So I set out to design a portal API that used his library. It has been only on free nights and weekends, where my brain is mush, but the structure is slowly starting to form.

In early 2024, Alfie reached out to me if I wanted to collaborate, and we did. We joined with Martin Sirringhaus, who worked on the WebAuthn authenticator library used in Firefox and is being sponsored by SUSE to work on this project, which is hosted at https://github.com/linux-credentials.

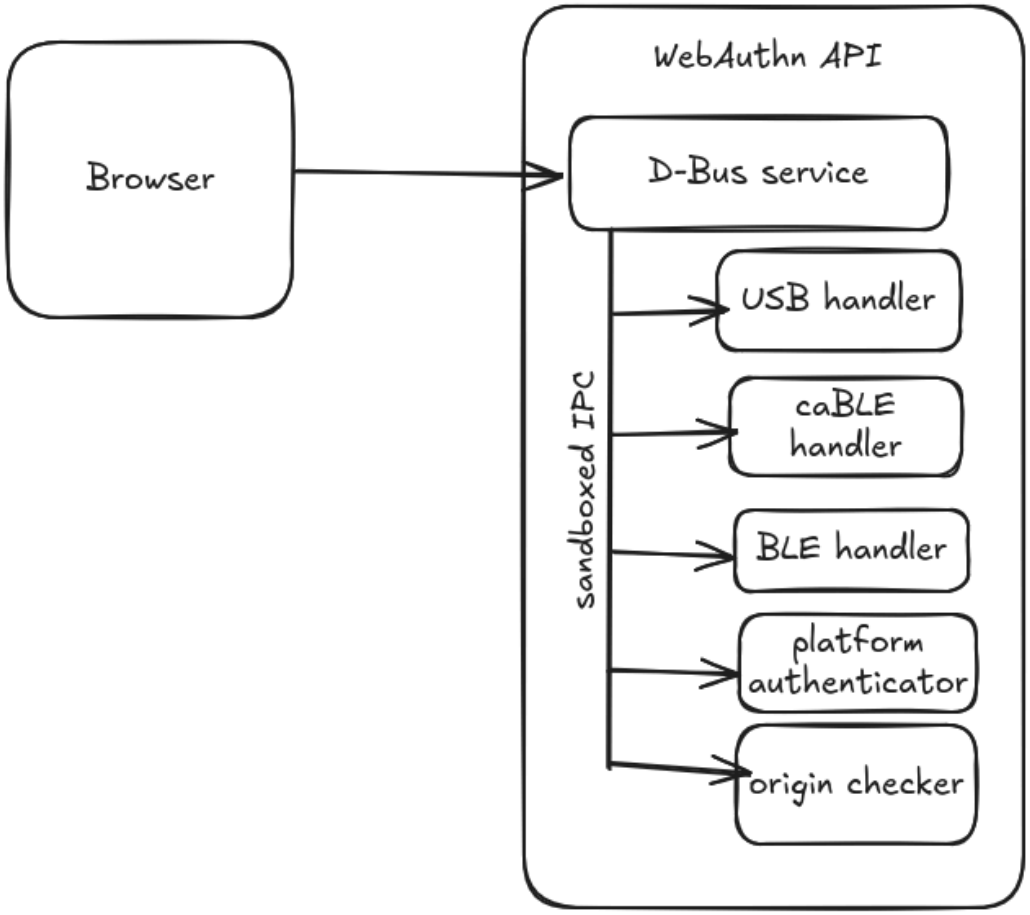
(We had hoped to make some more progress more quickly on the portal with funding from an open source grant, but unfortunately, the funding fell through. If you know of anyone willing to sponsor development on this work, let me know!)

Architecture proposal

Here are some features we want to build:

- USB authenticator support (done) ✅
- caBLE/hybrid authenticator support (in-progress)
- native Firefox integration
- An API for third-party passkey providers to hook into the passkey selection UI, similar to the settings available in Android, Apple, and soon in Windows.
- A platform authenticator

So far, we have wired up USB authenticator support and are in process of adding hybrid/caBLE support. (Shhh! Don't tell any, but we are testing this integration in Firefox with a web extension 🤫. We'll have a native Firefox integration by the end of this.) After that, we will work on productionizing the service, splitting it into multiple components connected via D-Bus IPC, each of which are sandboxed using systemd and Landlock security features. We also will provide SELinux/AppArmor policies to protect credential and configuration files needed to run the service.



A picture of IPC architecure of Linux WebAuthn Platform API. A browser or app calls the main API process over D-Bus, which handles different parts of the request by passing data to sandboxed child components.

You may ask, "Why split this up into so many processes?" There are a two main reasons: First, to contain vulnerabilities. Processes provide address space boundaries. Address space protections, SELinux, AppArmor and Landlock LSMs all work at the process boundary. Splitting the main D-Bus service, platform authenticator, authenticator I/O, etc. into separate processes allows us to give each of them the least privilege necessary. For example, if the USB I/O handler crashes due to a malicious or buggy authenticator, it can just die and be restarted, and it couldn't be used to read the secrets used by the platform authenticator.

The second reason is that it makes it easier to execute off some of the functionality needed for the platform authenticator inside a TEE, which I will explain in the kernel protection section below.

Implementation challenges/needs

Origin Checking, or lack thereof

I've been mentioning origin checking as an important authorization policy that provides phishing resistance for WebAuthn credentials. The other implementations (Apple and Android OSes) rely heavily on being able to evaluate application identity at runtime, which they can do because pretty much all user-level software is vetted to some degree, signed and verified. Unfortunately, I can't think of a good way to do this with Linux desktop OS capabilities. A recent comment on Lobsters stated this well:

> One of the problems with today's computers is that they are using abstractions designed for coarse-grained isolation of users to enforce fine-grained isolation of things owned by one user. I do want an environment where everything from the kernel on up can be modified by *the user*, but not one where it can be modified by *any piece of software that the user starts*.

Linux has very little to no code signing, at least not in a way that is available to the kernel so we can make policy decisions on it. However, Linux distributions are largely based on package managers, which check the integrity of packages on install. This means that the user is already trusting the official repos configured in their package manager: why not reuse those same signatures at runtime?

I think we're on the way: the upcoming Integrity Digest Cache (formerly DIGLIM) feature in the Linux kernel provides a way for package managers to integrate package signatures into LSM policies. This gets us part of the way there. A naive example of how this could be used is: when the platform API receives a request, it can use the D-Bus caller PID to find its executable, verify that its signature is valid according to the package manager [4], and then look up the package name of the executable, and use the package name somehow for whitelisting certain applications or mapping them to origins. This, of course, would only

work for packaged applications; we'd need to either choose not to support unpackaged applications (which might be OK), or store some sort of hardcoded exception list in a config database (with some sort of tamper protections).

Something similar has been done at Google on their Debian-based devices. One issue they ran into is that, unlike RPMs, there is not currently a place in DEB packages to store their signature metadata. So this would require some more changes to the ecosystem before we could make this widely available.

Sandboxed app runtimes, like Flatpak and Snaps, do typically provide application IDs, and there is even some work to put these application IDs in a standard place where D-Bus can access it. This doesn't completely solve the problem, since it doesn't address unsandboxed applications, which I think will be around for quite a while.

In the meantime, I'll probably wind up releasing an initial version of the platform API without origin checking but make sure to tell developers that they should use the API in such a way that when origin checking is later enabled that it doesn't break their code.

Hardening Platform Authenticator Credential Access

Our goal is to provide a platform authenticator that creates device-bound passkeys using a TPM. But, as we mentioned above, we should also make sure that the platform API has *exclusive* access to the credentials. By default, any process (including any user on any OS running on the machine) on the device can ask the TPM to sign data with any credential blob it has access to. That means that an attacker must gain access to the device (physically, or a malicious app or remote code execution vulnerability) to sign that, but then any process with access to the TPM key file could sign arbitrary WebAuthn challenges. No bueno.

Protecting from the user

We should at least make it so that arbitrary processes running as the logged-in user cannot access the key blobs. We can accomplish this by running the platform authenticator as a separate service user and having it own the files. This means that exploiting the keys requires privilege escalation.

Protecting from root

For extra protection, we should also add a MAC rule (SELinux/AppArmor) that only the platform authenticator service can access the TPM key blobs, so that even `root` processes cannot access the service. Of course, that only holds if the LSM policy is correct, which `root` can change. This can still be effective if the malicious `root` process runs in a sandbox (e.g. with seccomp) that disallows LSM policy changes, so it's still worth doing.

Protecting from the kernel

But what if we wanted to harden this even further: what if we wanted to protect the credentials, even if the OS itself is compromised? That would require some sort of access control at a level above the kernel, which is hardware in a typical scenario.

Apple's Approach

Apple devices have all the software and hardware made by the same vendor, so the OS can depend on specific hardware setups, like the Secure Enclave that can handle keys securely. Application signing requirements also allow the OS to restrict keys to a particular application or set of applications.

Windows' approach

The Linux ecosystem is more like Windows', with heterogeneous hardware and little to no signing of software. So how does Windows protect credentials from the kernel?

Windows has a feature, Virtualization-Based Security (VBS), that protects certain data. It's based on a Hyper-V feature called *virtual secure mode (VSM)*, where the OS is split into two or more VMs called *virtual trust levels (VTL)*. The VTLs are arranged in priority, so that VTL1 is more privileged than VTL0, for example. At boot, the bootloader starts the VTLs and boots the guest OS, the main OS the user interacts with, into VTL0, and boots a stripped-down secure OS into VTL1.

So far, this sounds like normal virtualization. But VSM also defines "hypercalls" (like syscalls from userspace to the kernel above it, but instead from the kernel to the hypervisor above it), so that the guest OS can communicate with the secure OS. The secure OS can communicate with the guest OS by sharing memory, so there is bidirectional communication. Because the hypercalls are very limited, and the amount of code running the secure OS is minimal, we get a much smaller trusted computing base (TCB) whose memory and CPU state cannot be corrupted by malicious processes in the guest OS, whether userspace or kernel processes (at least not without a hypervisor escape).

This basically gives Windows a TEE on any processor that supports the required virtualization features (basically all consumer processors released in the last decade).

Linux's approach?

How does Hyper-V VSM help us on Linux? Well, Hyper-V is actually both an implementation and specification, and there are two projects that are attempting to implement the Hyper-V

VSM spec for Linux: Heki, from Microsoft, and another one from a team at Amazon. These implementations are still in progress, but look very promising.

What would we still need to do when these are finished? I'm hitting the limits of my research on these topics, so it's a little fuzzy. But I think we would need to:

- Pick an OS for the secure OS, like OP-TEE[5]. There seems to be an implementation for Intel (x86 in general?); perhaps we'd just need to adapt the HAL to be able to use the VTL exits and interrupts and shared memory coming from VSM.
- For handling cryptographic secrets in a way that is unaccessible to the guest OS, we'd need to create a way to provision the secrets only to VTL1. Depending on how we boot VTL1, this could perhaps be done using systemd-boot. It stores the "boot phase" in a TPM PCR. We could generate and bind the secure OS's root secret to the `enter-initrd` phase. During boot, we'd start the secure OS with the root secret, change the PCR to `leave-initrd`, and then continue booting the guest OS. The root secret key blob would only be available in the secure OS's memory.
- Implement the platform authenticator as a Trusted Application (TA) inside the TEE using the GlobalPlatform TEE Internal Core API (which is implemented by OP-TEE). WebAuthn credentials created by the platform authenticator TA would be wrapped by the root secret provisioned to the secure OS.

At this point, we'd have WebAuthn credentials that are inaccessible to the kernel!

With the secure OS primitive, we may also be able to get value out of using Microsoft's SDCP protocol to securely connect to biometric sensors, which could then allow us to release WebAuthn credentials using biometrics instead of just a PIN. I am not aware of any Linux drivers for these that implement SDCP, but at least some reverse engineering has been done on some SDCP-supported devices, so the community may be able to continue that work.

Like I said, I need to do a bit more research to know if these are viable. But these ideas are not new: for example, Matthew Garrett has written before about virtualized security for Linux WebAuthn and biometrics and hiding TPM secrets from the kernel.

Other hopes and dreams

This is already too long, so I'm just going to throw a laundry list of other things I'd like to see in this space:

- FIDO2 credential management, so you can set up PINs and policies for new security keys, delete credentials from them, etc.
- Have the platform API mediate access to passwords along with passkeys. Passwords are going to be around for a long time, and while password manager extensions use,

they are hard to implement securely. The API wouldn't store passwords directly though, it'd probably just providing hooks for password manager applications.
- Similarly, having a TOTP API to store and autofill TOTPs automatically would be useful.
- Could we have some sort of autofill service and UI integration (like a GTK widget) injects third party
- We need a way to bind apps/packages to web origins, the equivalent of Related Origins, Digital Asset Links, and Apple Site Associations. Could we use something like the App binding mechanism: Web Manifest spec?
- A privacy-preserving proxy service for looking up origins related to an rpId or app ID.
- On-device queryable signatures in DEB packages.
- An open-source implementation of caBLE/hybrid server for debugging, potentially provided as a service for Linux-based phones, like the PinePhone or Librem.

Thanks for reading! If you've made it this far, you may be interested in helping out. 🙂 Feel free to reach out!

Resources

- Enabling Windows Credential Guard in KVM (KVM Forum, YouTube)
- Emulating Hyper-V's Virtual Secure Mode (VSM) with QEMU (YouTube)
- Credential Guard Overview (learn.microsoft.com)
- Virtualization Based Security Overview (learn.microsoft.com)
- Virtual Secure Mode (VSM) (learn.microsoft.com)
- On-device WebAuthn and What Makes It Hard to Do Well
- Why does GNOME Fingerprint Not Unlock the Keyring?
- DIGLIM overview, predecessor to Integrity Digest Cache

[1] This has implications for account recovery, since if the credentials are tied to the device, if you lose the device, you're locked out. Because of this, the major platforms have all implemented passkey syncing within their ecosystem: Apple devices sync via iCloud Keychain, Android devices sync via Google Password Manager, and as of recently, Windows syncs to Microsoft Account. There is standardization work to make passkeys sync across these ecosystems as well.

[2] If you're curious about device support, you can check out the awesome compatibility matrix on passkeys.dev made by Tim Cappali and others.

[3] One downside of putting the UI in the platform, although the UI between browsers/apps on a single device is consistent, it is inconsistent between devices on different platforms. When browsers controlled the whole UI, it made it consistent across different platforms. I think this is a fine tradeoff based on my (perhaps incorrect) intuition that since Apple is basically the only vendor that provides both desktop/laptop and mobile devices, consumers either will tend to stay within a single platform for their devices (all Apple), or would have already have had a mix of devices (Windows laptop + Android or iPhone, or

Mac + Android) and therefore would be used to differing platforms anyway. (This concession does not apply to syncing passkeys, just the UI; I still think we should work to accomplish usable cross-ecosystem syncing in native platforms.)

[4] I believe that Integrity Digest Cache and IMA take care of TOCTOU problem of checking the race condition of checking the signature of the process as it was when the process started vs. what it is on disk at the time of the WebAuthn request.

[5] I've also glanced at sel4, which is interesting due to its formal verification. Also interesting, both sel4 and sepOS, Apple's secure OS for the Secure Enclave, happen to be based on the L4 microkernel.

[More posts]