WireGuard Pro is the 100% rootless VPN-in-a-container solution that I developed using Cloudflare’s user space Rust implementation of the WireGuard protocol, BoringTun, and the small UI I created for managing clients and viewing statistics. There were two primary factors that motivated me to develop WireGuard Pro.

Firstly, I needed a VPN to secure my internet connection from unprotected public networks. However, I did not want to pay for a service since I have a small AWS Lightsail VM that I use as a home lab. Once I discovered that self-hosting a VPN is actually not all that difficult, I went down the rabbit hole of comparing different self-hosted VPN options. I finally settled on the WireGuard protocol since it has a minimal codebase and is very modern and cutting-edge in the way it handles cryptographic primitives compared to other self hosted solutions, which in conjunction makes it significantly faster than other self-hosted solutions. Once I decided on WireGuard as the VPN protocol, I then came to my first major blocker. I have a strong preference to run applications rootlessly and inside a container for both isolation and security purposes, but rootless containerization doesn’t play well with the kernel (C) implementation of WireGuard.

Luckily, I am not the only person that prefers rootlessness whenever possible, and two viable alternatives to the kernel version of WireGuard exist: WireGuard-Go and BoringTun. As the name implies, WireGuard-Go is implemented with the Go programming language, and was the first user space-compatible version of WireGuard. The Go implementation is notably slower than the Kernel implementation, but for awhile was the only option for anyone who wanted to run WireGuard in a nonstandard environment of any sort. While Go does have low level capabilities, due to its garbage collected environment and goroutine scheduler, it simply cannot compete speed-wise with C or C++. For a more performant implementation than go, there needs to be no cost to abstraction and there should not be a garbage collector, so naturally, Rust would be the best choice for the two aforementioned reasons.
Cloudflare‘s open source arm took it upon themselves to develop the Rust-based BoringTun, which is similar to WireGuard-Go in that it can run rootlessly, but superior due to Rust simply being a more performant language than Go.

Now that I had decided on BoringTun for the implementation, I started trying to shim BoringTun into a few existing containerized WireGuard UIs to no avail. The prebuilt options unfortunately ended up requiring so much modification to accommodate BoringTun versus Kernel/Go that I came to my second motivation for creating WireGuard Pro: no existing user interface was minimal enough, functional enough, or flexible enough to accommodate my needs, so I should create my own version, since spinning up a small Flask API and plain old html-css-js UI does not require all that much work.

Since WireGuard Pro is a 100% rootless container, my preferred way of running it is via a Podman systemd .container quadlet. Podman integrates with systemd quite nicely and will generate a daemon service based on a .container quadlet to run the container in the background, with systemd also automatically starting the service on system boot as well.

In addition to the container quadlet, I also prefer to deploy it behind a reverse proxy with Caddy, which I favor over options like Nginx for its ease of configuration and automated SSL certification process. Deploying this process requires very limited input from a root or admin user on the system that it runs on. By default, rootless Podman won't allow applications to run on ports lower than 1000, but in order to avoid issues with https, I allow the rootless Podman user to bind to privileged ports, starting with 80 to protect low ports such as 22 for ssh, which is a very limited and calculated allowance, as routing https to non standard ports adds unnecessary complexity for no functional security advantage. Additionally, the user will need to be allowed to linger so that the process does not exit on logout. Deployment behind a reverse proxy is not necessary, end users could theoretically use a different method if they so choose, but the reverse proxy is advised for ease of setup. Enabling loginctl lingering is necessary though, as without that set the user-level service will terminate on logout.

Inside the container, we rely on a few additional tricks, enhancements, and tools that differ standard container networking. Firstly, to handle WireGuard's UDP packets, I employed slirp4netns to emulate root space and facilitate safe arrival of packets into rootless container networking safe egress back to the open internet. While slirp4netns is not the most performant networking solution for containers, it is the only networking method or tool that works properly with BoringTun, as it does not yet support the superior socket activation and proxy method employed by my UI. Socket Activation is the only way that a non root user can achieve rootful bare metal-levels of network throughput. With a simple proxy like caddy, we can proxy the socket to a local subnet, which is far superior than forwarding ports from the container.

While BoringTun is the best choice for a rootless VPN, it does still have some shortcomings. Firstly, when BoringTun is used on a small VM such as my AWS Lightsail instance, users will run into a throughput bottleneck on encryption and decryption times, as the single thread performance is not ideal in these environments. Additionally, as previously stated, BoringTun does not naturally accept sockets for socket activation. To resolve these issues, I plan to either make the following contributions to BoringTun or fork my own implementation with the following modifications.

First things first, I really need to add support for socket activation, as it is the best way to handle rootless container networking, and allows for native bare metal levels of throughput. Next, I want to batch crypto calls so that they sit in a buffer instead of being processed packet by packet, which feels more in line with the nature of UDP communication. Furthermore, I can batch reads and writes to the tunnel interface so that the process includes a buffer just like the crypto calls for a more UDP-like approach. Beyond that, even simply pre allocating a set of common buffers, would speed things up a lot since it would eliminate the creation step whenever a buffer is used.

While I have yet to achieve my desired throughput rootlessly at the current time, I believe with the adjustments to BoringTun that I proposed above, I can achieve my rootful-equivalent throughput from user space. Furthermore, I believe that I can do a bit more on the network security side beyond scheduling auto expiration for clients such as rotating the external IP address on a rolling basis, integrating OAuth support, and integrations for other popular home networking tools such as Mullvad and Pihole. Additionally, in the UI I could add more monitoring components, even utilizing tools like Prometheus or Garfana. In conclusion, I believe that I have developed a strong launch point for my rootless VPN project WireGuard Pro, and that WireGuard Pro could end up becoming a serious open source project for me to manage.