WireGuard® is an extremely simple yet fast and modern VPN that utilizes **state-of-the-art cryptography (protocol/)**. It aims to be faster (performance/), simpler (quickstart/), leaner, and more useful than IPSec, while avoiding the massive headache. It intends to be considerably more performant than OpenVPN. WireGuard is designed as a general purpose VPN for running on embedded interfaces and super computers alike, fit for many different circumstances. Initially released for the Linux kernel, it is now cross-platform and widely deployable. It is currently under heavy development, but already it might be regarded as the most secure, easiest to use, and simplest VPN solution in the industry.

Simple & Easy-to-use

WireGuard aims to be as easy to configure and deploy as SSH. A VPN connection is made simply by exchanging very simple public keys – exactly like exchanging SSH keys – and all the rest is transparently handled by WireGuard. It is even capable of roaming between IP addresses, just like Mosh (http://mosh.mit.edu/). There is no need to manage connections, be concerned about state, manage daemons, or worry about what's under the hood. WireGuard presents an extremely basic yet powerful interface.

Cryptographically Sound

WireGuard uses state-of-the-art cryptography, like the Noise protocol framework (http://www.noiseprotocol.org), Curve25519 (http://cr.yp.to/ecdh.html), ChaCha20 (http://cr.yp.to/chacha.html), Poly1305 (http://cr.yp.to/mac.html), BLAKE2 (https://blake2.net/), SipHash24 (https://131002.net/siphash/), HKDF (https://eprint.iacr.org/2010/264), and secure trusted constructions. It makes conservative and reasonable choices and has been reviewed by cryptographers.

</> Minimal Attack Surface

WireGuard has been designed with ease-of-implementation and simplicity in mind. It is meant to be easily implemented in very few lines of code, and easily auditable for security vulnerabilities. Compared to behemoths like *Swan/IPsec or OpenVPN/OpenSSL, in which auditing the gigantic

codebases is an overwhelming task even for large teams of security experts, WireGuard is meant to be comprehensively reviewable by single individuals.

ϟ High Performance

A combination of extremely high-speed cryptographic primitives and the fact that WireGuard lives inside the Linux kernel means that secure networking can be very high-speed. It is suitable for both small embedded devices like smartphones and fully loaded backbone routers.

🎓 Well Defined & Thoroughly Considered

WireGuard is the result of a lengthy and thoroughly considered academic process, resulting in the technical whitepaper (papers/wireguard.pdf), an academic research paper which clearly defines the protocol and the intense considerations that went into each decision.

Conceptual Overview

If you'd like a general conceptual overview of what WireGuard is about, read onward here. You then may progress to installation (install/) and reading the quickstart instructions (quickstart/) on how to use it.

If you're interested in the internal inner workings, you might be interested in the brief summary of the protocol (protocol/), or go more in depth by reading the technical whitepaper (./papers/wireguard.pdf), which goes into more detail on the protocol, cryptography, and fundamentals. If you intend to implement WireGuard for a new platform, please read the cross-platform notes (xplatform/).

WireGuard securely encapsulates IP packets over UDP. You add a WireGuard interface, configure it with your private key and your peers' public keys, and then you send packets across it. All issues of key distribution and pushed configurations are *out of scope* of WireGuard; these are issues much better left for other layers, lest we end up with the bloat of IKE or OpenVPN. In contrast, it more mimics the model of SSH and Mosh; both parties have each other's public keys, and then they're simply able to begin exchanging packets through the interface.

Simple Network Interface

WireGuard works by adding a network interface (or multiple), like `eth0` or `wlan0`, called `wg0` (or `wg1`, `wg2`, `wg3`, etc). This network interface can then be configured normally using `ifconfig(8)` or `ip-address(8)`, with routes for it added and removed using `route(8)` or `ip-route(8)`, and so on with all the ordinary networking utilities. The specific WireGuard aspects of the interface are configured using the `wg(8)` (https://git.zx2c4.com/WireGuard/about/src/tools/man/wg.8) tool. This interface acts as a tunnel interface.

WireGuard associates tunnel IP addresses with public keys and remote endpoints. When the interface

sends a packet to a peer, it does the following:

1. This packet is meant for 192.168.30.8. Which peer is that? Let me look... Okay, it's for peer `ABCDEFGH`. (Or if it's not for any configured peer, drop the packet.)
2. Encrypt entire IP packet using peer `ABCDEFGH`'s public key.
3. What is the remote endpoint of peer `ABCDEFGH`? Let me look... Okay, the endpoint is UDP port 53133 on host 216.58.211.110.
4. Send encrypted bytes from step 2 over the Internet to 216.58.211.110:53133 using UDP.

When the interface receives a packet, this happens:

1. I just got a packet from UDP port 7361 on host 98.139.183.24. Let's decrypt it!
2. It decrypted and authenticated properly for peer `LMNOPQRS`. Okay, let's remember that peer `LMNOPQRS`'s most recent Internet endpoint is 98.139.183.24:7361 using UDP.
3. Once decrypted, the plain-text packet is from 192.168.43.89. Is peer `LMNOPQRS` allowed to be sending us packets as 192.168.43.89?
4. If so, accept the packet on the interface. If not, drop it.

Behind the scenes there is much happening to provide proper privacy, authenticity, and perfect forward secrecy, using state-of-the-art cryptography.

Cryptokey Routing

At the heart of WireGuard is a concept called *Cryptokey Routing*, which works by associating public keys with a list of tunnel IP addresses that are allowed inside the tunnel. Each network interface has a private key and a list of peers. Each peer has a public key. Public keys are short and simple, and are used by peers to authenticate each other. They can be passed around for use in configuration files by any out-of-band method, similar to how one might send their SSH public key to a friend for access to a shell server.

For example, a server computer might have this configuration:

```
[Interface]
PrivateKey = yAnz5TF+lXXJte14tji3zlMNq+hd2rYUIgJBgB3fBmk=
ListenPort = 51820

[Peer]
PublicKey = xTIBA5rboUvnH4htodjb6e697QjLERt1NAB4mZqp8Dg=
AllowedIPs = 10.192.122.3/32, 10.192.124.1/24

[Peer]
PublicKey = TrMvSoP4jYQlY6RIzBgbssQqY3vxI2Pi+y71lOWWXX0=
AllowedIPs = 10.192.122.4/32, 192.168.0.0/16

[Peer]
PublicKey = gN65BkIKy1eCE9pP1wdc8ROUtkHLF2PfAqYdyYBz6EA=
AllowedIPs = 10.10.10.230/32
```

And a client computer might have this simpler configuration:

```
[Interface]
PrivateKey = gI6EdUSYvn8ugXOt8QQD6Yc+JyiZxIhp3GInSWRfWGE=
ListenPort = 21841

[Peer]
PublicKey = HIgo9xNzJMWLKASShiTqIybxZ0U3wGLiUeJ1PKf8ykw=
Endpoint = 192.95.5.69:51820
AllowedIPs = 0.0.0.0/0
```

In the server configuration, each peer (a client) will be able to send packets to the network interface with a source IP matching his corresponding list of allowed IPs. For example, when a packet is received by the server from peer `gN65BkIK...`, after being decrypted and authenticated, if its source IP is 10.10.10.230, then it's allowed onto the interface; otherwise it's dropped.

In the server configuration, when the network interface wants to send a packet to a peer (a client), it looks at that packet's destination IP and compares it to each peer's list of allowed IPs to see which peer to send it to. For example, if the network interface is asked to send a packet with a destination IP of 10.10.10.230, it will encrypt it using the public key of peer `gN65BkIK...`, and then send it to that peer's most recent Internet endpoint.

In the client configuration, its single peer (the server) will be able to send packets to the network interface with *any* source IP (since 0.0.0.0/0 is a wildcard). For example, when a packet is received from peer `HIgo9xNz...`, if it decrypts and authenticates correctly, with any source IP, then it's allowed onto the interface; otherwise it's dropped.

In the client configuration, when the network interface wants to send a packet to its single peer (the server), it will encrypt packets for the single peer with *any* destination IP address (since 0.0.0.0/0 is a wildcard). For example, if the network interface is asked to send a packet with any destination IP, it will encrypt it using the public key of the single peer `HIgo9xNz...`, and then send it to the single peer's most recent Internet endpoint.

In other words, when sending packets, the list of allowed IPs behaves as a sort of routing table, and when receiving packets, the list of allowed IPs behaves as a sort of access control list.

This is what we call a *Cryptokey Routing Table*: the simple association of public keys and allowed IPs.

Any combination of IPv4 and IPv6 can be used, for any of the fields. WireGuard is fully capable of encapsulating one inside the other if necessary.

Because all packets sent on the WireGuard interface are encrypted and authenticated, and because there is such a tight coupling between the identity of a peer and the allowed IP address of a peer, system administrators do not need complicated firewall extensions, such as in the case of IPSec, but rather they can simply match on "is it from this IP? on this interface?", and be assured that it is a secure and authentic packet. This greatly simplifies network management and access control, and provides a great deal more assurance that your iptables rules are actually doing what you intended for them to do.

Built-in Roaming

The client configuration contains an *initial* endpoint of its single peer (the server), so that it knows where to

send encrypted data before it has received encrypted data. The server configuration doesn't have any initial endpoints of its peers (the clients). This is because the server discovers the endpoint of its peers by examining from where correctly authenticated data originates. If the server itself changes its own endpoint, and sends data to the clients, the clients will discover the new server endpoint and update the configuration just the same. Both client and server send encrypted data to the most recent IP endpoint for which they authentically decrypted data. Thus, there is full IP roaming on both ends.

Ready for Containers

WireGuard sends and receives encrypted packets using the network namespace in which the WireGuard interface was originally created (netns/). This means that you can create the WireGuard interface in your main network namespace, which has access to the Internet, and then move it into a network namespace belonging to a Docker container as that container's *only* interface. This ensures that the only possible way that container is able to access the network is through a secure encrypted WireGuard tunnel.

Learning More

Consider glancing at the commands & quick start (quickstart/) for a good idea of how WireGuard is used in practice. There is also a description of the protocol, cryptography, & key exchange (protocol/), in addition to the technical whitepaper (./papers/wireguard.pdf), which provides the most detail.

About The Project

Work in Progress

WireGuard is not yet complete. *You should not rely on this code.* It has not undergone proper degrees of security auditing and the protocol is still subject to change. We're working toward a stable 1.0 release, but that time has not yet come. There are experimental snapshots tagged with "0.0.YYYYMMDD", but these should not be considered real releases and they may contain security vulnerabilities (which would *not* be eligible for CVEs, since this is pre-release snapshot software). If you are packaging WireGuard, you *must* keep up to date with the snapshots.

However, if you're interested in helping out, we could really use your help and we readily welcome any form of feedback and review. There's currently quite a bit of work to do on the project todo list (todo/), and the more folks testing this out, the better.

Contributing

Get involved in the WireGuard development discussion by joining the mailing list (https://lists.zx2c4.com/mailman/listinfo/wireguard). This is where all development activities occur. Submit patches using `git-send-email`, similar to the style of LKML. You may also discuss development related activity on `#wireguard` on Freenode.

Contact the Team

All general questions and contributions should go to the mailing list (https://lists.zx2c4.com/mailman/listinfo/wireguard), but if you'd like to contact us privately for a particular reason, you may reach us at team@wireguard.com (mailto:team@wireguard.com).

Please report any security issues to security@wireguard.com (mailto:security@wireguard.com). You may encrypt your security-related emails using GPG key `20A749FC7012A5DE03AE` (https://www.zx2c4.com/keys/AB9942E6D4A4CFC3412620A749FC7012A5DE03AE.asc).

Source Code

WireGuard is hosted in the ZX2C4 Git Repository (https://git.zx2c4.com/WireGuard/). You may clone the repository via:

```
$ git clone https://git.zx2c4.com/WireGuard
```

or

```
$ git clone git://git.zx2c4.com/WireGuard
```

Alternatively, if you have push access, you may clone via SSH:

```
$ git clone ssh://git@git.zx2c4.com/WireGuard
```

License

This project is released under the GPLv2 (https://git.zx2c4.com/WireGuard/about/COPYING).

Tweet about this page! (https://twitter.com/share)

© Copyright 2015-2018 Jason A. Donenfeld. All Rights Reserved. "WireGuard" and the "WireGuard" logo are registered trademarks of Jason A. Donenfeld.

This project is from ZX2C4 (https://www.zx2c4.com/) and from Edge Security (https://www.edgesecurity.com/), a firm devoted to information security research expertise.

(https://www.edgesecurity.com/)