Corsi Linux Avanzati 2019

WireGuard (VPN)

POLITECNICO OPEN

unix LABS

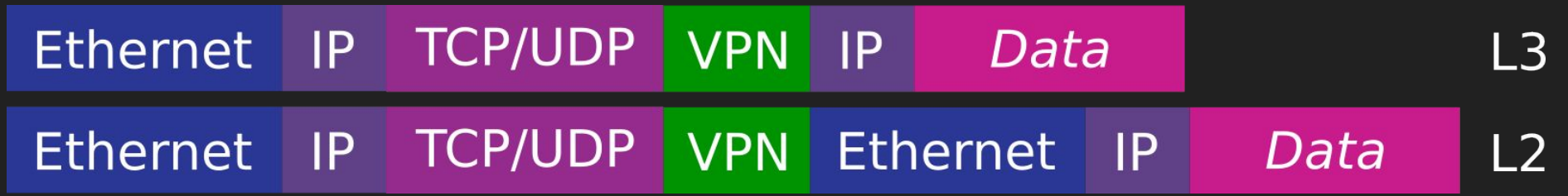
Davide Depau <davide@depau.eu>

What is a Virtual Private Network?

- It's an infrastructure that extends a private network across a public network
- Devices connected to a VPN can contact the other nodes as if they were directly connected to them

Virtual Private Network

A VPN can work either in **layer 2** or in **layer 3**



On Linux

- Virtual network interfaces are used to create VPN tunnels
- **tun** interfaces provide **layer 3 tunnelling**
- **tap** interfaces provide **layer 2 tunnelling**
- We won't see how to create them as most VPN implementations will create their interfaces on their own

WireGuard

- **Layer 3** VPN
- Extremely fast
  - Implemented as a **Linux kernel module** ⇒ as fast as it can be
  - Can be used in **userspace** if a kernel module is not desired
  - Almost stateless
  - Very simple protocol
- Highly reliable
  - Built-in roaming support
  - Created with DDoS attack resistance in mind

Use cases

- Limitation: Layer 3 ⇒ **can't be bridged** to other network interfaces
  - You need a different LAN with optional static routes
- Personal VPN: perfect
  - It allows roaming throughout different interfaces, downloads don't break if, for example, you quickly switch from mobile to Wi-Fi
  - Low overhead
  - Clients for every mobile platform - on rooted Android devices with custom kernel it can be used natively
- Connecting servers among cloud providers: awesome
  - Connection is secure, packets are authenticated
  - Allowed IPs setting is an additional layer of security
  - Connection resumes automatically if a link goes down then back up
- Linux Containers
  - We'll see that later

How it works

1. The WireGuard implementation (kernel module or wireguard-go) provides a tunnel interface
2. The interface is given an IP address (with the ip tool, or systemd-networkd)
3. The wg tool is used to set the WireGuard-specific parameters
4. Optionally, routing rules are added

Routing

- Package on regular Ethernet/Wi-Fi private networks are sent to the right place thanks to ARP - Address Resolution Protocol
  - The device asks the network for the MAC address matching the IP address it wants to contact
  - The host in question replies with the requested info

Routing

However, WireGuard works in layer 3: no MAC addresses, no ARP.

How does WireGuard know what to do?

Cryptokey routing

Cryptokey routing

- Every peer is identified by
  - A public key
  - Its allowed IP addresses (a list of IP addresses with their netmasks)
  - Its endpoint public IP address:port (which can be dynamic for “clients”)
- The public/private key pair can be generated using the wg tool
- WireGuard supports both IPv4 and IPv6, for both the “public” peer addressing and the internal VPN addresses
- It also supports making IPv6 travel over IPv4 and vice versa

Cryptokey routing

When a packet is being routed (**sent**)

1. Based on its routing table, the OS picks the outgoing interface
2. WireGuard analyses the packet and it
    a. Identifies the destination peer based on the packet's destination address and the peers' AllowedIPs
    b. Encrypts the packet using the peer's public key and signs it with its own private key
    c. Sends it over UDP to the peer's last known public IP address (which may change over time)

Cryptokey routing

When a packet is **received**

1. The packet is decrypted and authenticated against the peer's public key
    a. If the peer's public IP address has changed, it is stored internally
2. The IP frame is analyzed
    a. The source IP address is verified against the list of AllowedIPs for the source peer
    b. If the peer is allowed to send packets from that address, it is routed, otherwise it is dropped

Cryptokey routing

This has some implications:

- Peers that are to be used as gateways must have AllowedIPs = 0.0.0.0/0 set in the clients peers configs, otherwise the clients would drop packets from the Internet coming from that peer.
- All packets are authenticated by WireGuard itself. This means that if a packet comes from a known IP address from the WireGuard interface, the packet can be assumed secure and authentic.

Roaming

- An initial external (public) endpoint must be specified for at least one node in every peer pair to bootstrap the connection
- Once encrypted and authenticated data is received from a (new) IP address, the peers learn that address and will use it to send data to that peer
- WireGuard is not very *chatty*
  - It almost always only communicates with the peers when data is actually sent
- Keepalive packets may optionally be enabled to help traverse NATs (it does not do NAT hole-punching, though)

How to set it up (manually)

All commands need to be run as root

1. Add a WireGuard network interface
   a. (Linux kernel module): `ip link add dev wg0 type wireguard`
   b. (userspace implementations): `wireguard-go wg0`
2. Set its IP address(es)
   a. (normal LAN) `ip address add dev wg0 192.168.2.1/24`
   b. (peer2peer) `ip address add dev wg0 192.168.2.1 peer 192.168.2.2`
3. Set its WireGuard-specific configuration
   a. `wg setconf wg0 myconfig.conf`

The config file (server 1)

```
[Interface]
Address = 192.168.42.1/24
PostUp = iptables -A FORWARD -i %i -j ACCEPT;
iptables -t nat -A POSTROUTING -o eth0 -j
MASQUERADE; iptables -A FORWARD -m conntrack
--ctstate RELATED,ESTABLISHED -j ACCEPT
PostDown = iptables -D FORWARD -i %i -j ACCEPT;
iptables -t nat -D POSTROUTING -o eth0 -j
MASQUERADE; iptables -D FORWARD -m conntrack
--ctstate RELATED,ESTABLISHED -j ACCEPT
ListenPort = 1194
PrivateKey = [server 1 private key]

[Peer]
PublicKey = [server 2 public key]
AllowedIPs = 192.168.42.2/32, 192.168.42.0/24
Endpoint = wgserver2.example.com:1194
PersistentKeepalive = 25

[Peer]
PublicKey = [client public key]
AllowedIPs = 192.168.42.100/32
```

- Setup with two peers with known external IPs (servers) and one peer with unknown external IP (client)
- This specific config file is for one of the servers and includes some iptables commands to (optionally) make it work as a NATting gateway
- The first server may reach the second one directly, because its endpoint is specified
- The client's endpoint will be discovered when the client sends authenticated data to this peer

The config file (server 2)

```
[Interface]
Address = 192.168.42.2/24
PostUp = iptables -A FORWARD -i %i -j ACCEPT;
iptables -t nat -A POSTROUTING -o eth0 -j
MASQUERADE; iptables -A FORWARD -m conntrack
--ctstate RELATED,ESTABLISHED -j ACCEPT
PostDown = iptables -D FORWARD -i %i -j ACCEPT;
iptables -t nat -D POSTROUTING -o eth0 -j
MASQUERADE; iptables -D FORWARD -m conntrack
--ctstate RELATED,ESTABLISHED -j ACCEPT
ListenPort = 1194
PrivateKey = [server 2 private key]

[Peer]
PublicKey = [server 1 public key]
AllowedIPs = 192.168.42.1/32, 192.168.42.0/24
Endpoint = wgserver1.example.com:1194

[Peer]
PublicKey = [client public key]
AllowedIPs = 192.168.42.100/32
```

- This config is for the second server and is basically the same as the first server
- The 2nd server may also reach the 1st directly
- The client will get to pick which one will be used as a gateway (or choose to use none of them); it will be able to reach them by their own address anyway

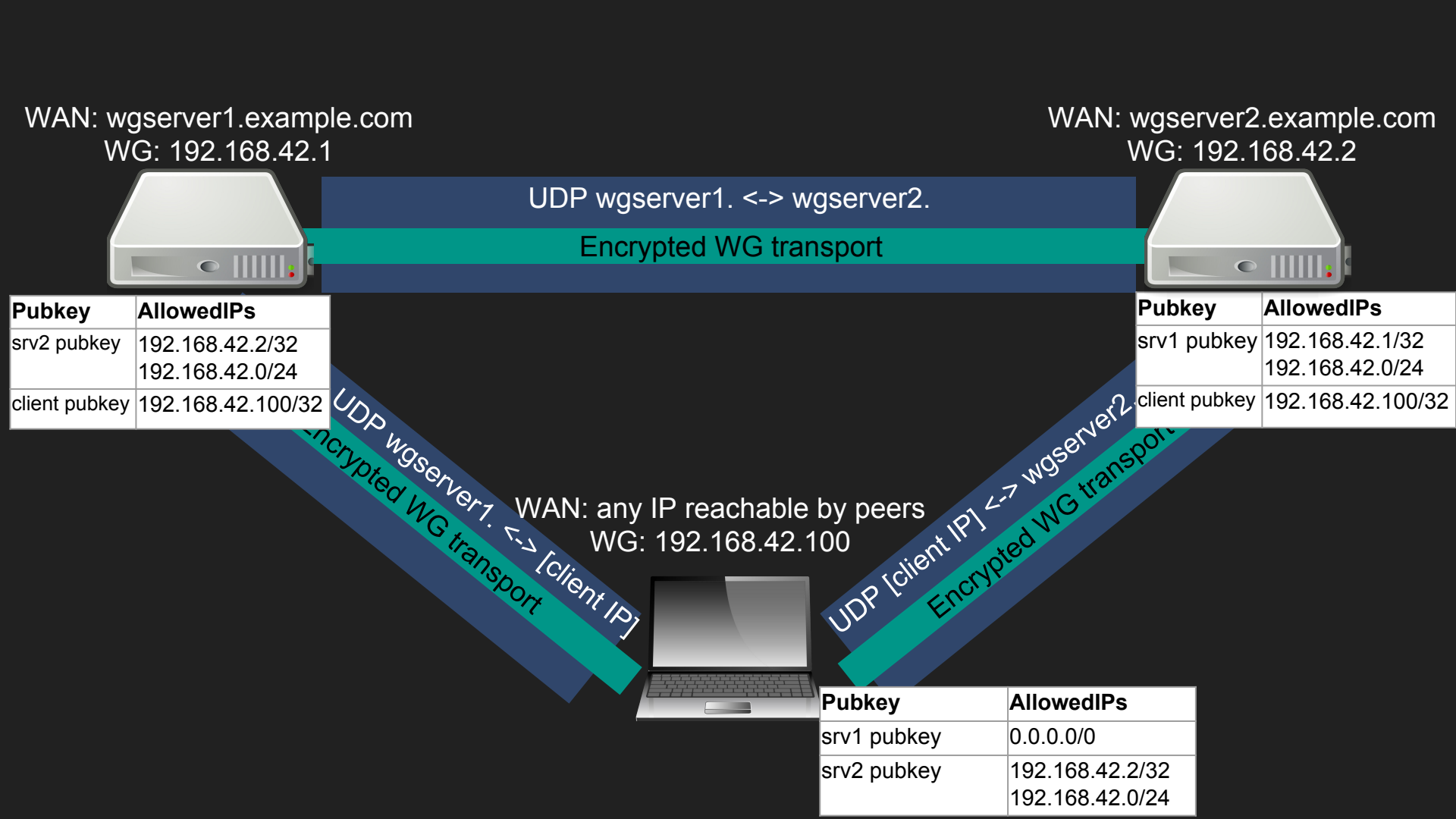
The config file (client)

```
[Interface]
Address = 192.168.42.100/24
DNS = 1.1.1.1
PrivateKey = [client private key]

[Peer]
PublicKey = [server 1 public key]
AllowedIPs = 0.0.0.0/0
Endpoint = wgserver1.example.com:1194

[Peer]
PublicKey = [server 2 public key]
AllowedIPs = 192.168.42.2/32, 192.168.42.0/24
Endpoint = wgserver2.example.com:1194
```

- The 1st server has AllowedIPs = 0.0.0.0/0. This means it will be allowed to send packets with any source IP address, i.e. it can be a gateway to the WAN
- The client will be able to reach both servers directly because both endpoints are specified
- The DNS is optional, of course

Some considerations

- PostUp/PostDown iptables commands are only needed to set up NAT on the servers so they can be used as gateways to the WAN.
  They’re not needed in a p2p/star network layout where each node will reach the WAN on its own and only needs WireGuard to communicate securely with its peers.
- AllowedIPs = 0.0.0.0/0 is only needed for the gateway setup. If every node needs to communicate only with its peers, only /32 AllowedIPs should be used for extra security.


WAN: wgserver1.example.com
WG: 192.168.42.1
WAN: wgserver2.example.com
WG: 192.168.42.2
UDP wgserver1. <-> wgserver2.
Encrypted WG transport
Pubkey
AllowedIPs
srv2 pubkey
192.168.42.2/32
192.168.42.0/24
client pubkey
192.168.42.100/32
Pubkey
AllowedIPs
srv1 pubkey
192.168.42.1/32
192.168.42.0/24
client pubkey
192.168.42.100/32
UDP wgserver1. <-> [client IP]
Encrypted WG transport
WAN: any IP reachable by peers
WG: 192.168.42.100
UDP [client IP] <-> wgserver2.
Encrypted WG transport
Pubkey
AllowedIPs
srv1 pubkey
0.0.0.0/0
srv2 pubkey
192.168.42.2/32
192.168.42.0/24

How to set it up (automatically)

- The config files can be placed in a standard location in each host (`/etc/wireguard/<configname>.conf`)
- wg-quick can then be used to set everything up
  - A WireGuard interface with the same name as the config file will be created
  - IP+netmask is automatically set from the config file, [Interface] → Address
  - DNS, if specified, is automatically written to /etc/resolv.conf
  - Entries are automatically added to the kernel routing table
  - {Pre,Post}{Up,Down} commands are executed
- Syntax: `wg-quick {up,down} <configname>`
- It will try to find a config named /etc/wireguard/*configname*.conf

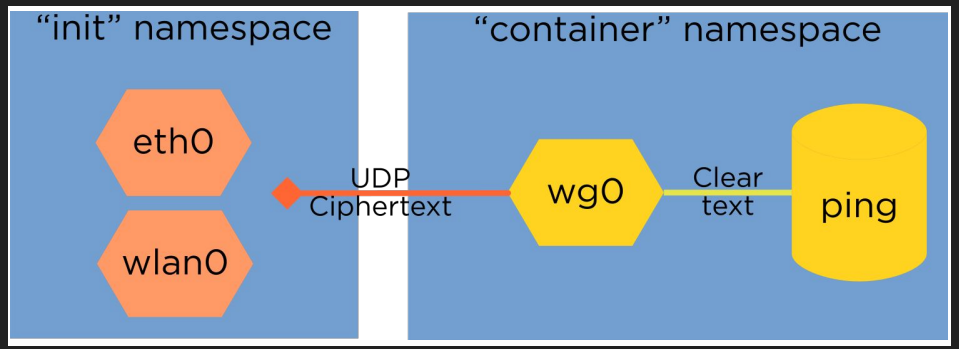
How to set it up (automatically)

```
[Unit]
Description=WireGuard via
wg-quick(8) for %I
After=network-online.target
Wants=network-online.target

[Service]
Type=oneshot
RemainAfterExit=yes
ExecStart=/usr/bin/wg-quick up %i
ExecStop=/usr/bin/wg-quick down %i

[Install]
WantedBy=multi-user.target
```

- wg-quick usually comes with a systemd service template that can be used to set up the interfaces automatically at boot
- It can be enabled on distros with systemd init system with `systemctl enable --now wg-quick@configname.service`

Containers

- The WireGuard interface *remembers* which network namespace it was created in
- It can then be moved to another namespace (i.e. a container namespace); the UDP socket will stay in the original namespace
- https://www.tauceti.blog/post/kubernetes-the-not-so-hard-way-with-ansible-wireguard/



Stability

- In general, WireGuard is considered unstable and the kernel module hasn’t been mainlined yet, though effort is being put in that direction
- The protocol itself may be subject to changes, so ensuring all nodes run the same version of WireGuard may be desired
- However, the authors say that mostly to “cover their back” in case anything breaks
- WireGuard has undergone all sorts of formal verification, covering both the protocol and cryptography
- Many people use it in production environments, including a bunch of commercial VPN providers

Performance

- Being in-kernel, simple, designed for parallelism and resistance to DoS, WireGuard offers incredible performance compared to most VPN solutions
- It appears *stateless* to userspace. Once you set it up you can forget about it, it will “just work”
- In my tests, the bandwidth has always almost matched the link’s bandwidth, usually with barely 2-5% overhead at most, even on hosts with RISC CPUs (ARM)
- Official benchmarks show it’s 4x faster than OpenVPN
- Userspace implementation obviously are a bit slower, but still faster than OpenVPN

Security

- Some aspects are considered controversial, in particular the fact that WireGuard implements cryptographic functions instead of using the kernel's Crypto API
  - This was done because the Crypto API wasn't as flexible as required and was not very fast
  - The cryptographic functions have undergone through formal verification, though, and are considered "secure"
- The fact that WireGuard runs in kernel space implies that any bugs may have some serious implications
  - However, the module has very little lines of code (~4000, vs. ~120000 + OpenSSL for OpenVPN) which can be reviewed even by individuals quite easily
  - Userspace implementations may be used if this is a concern

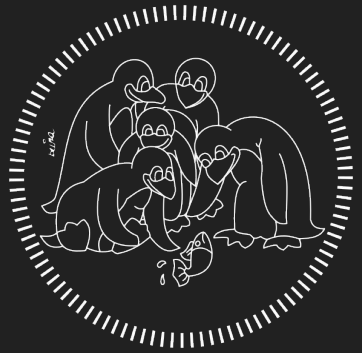
Security

- When configured incorrectly, instead of just working insecurely, WireGuard will simply refuse to work
- This may happen, for example, when peers have wrong public/private key pairs, AllowedIPs for a peer has been configured incorrectly

Links

- Linux Plumbers Conference 2018 slides: https://www.wireguard.com/talks/lpc2018-wireguard-slides.pdf
- The main website: https://www.wireguard.com/install/
- ArchWiki, as always: https://wiki.archlinux.org/index.php/WireGuard

Thank you!



Davide Depau <davide@depau.eu>