WireGuard

FAST, MODERN, SECURE VPN TUNNEL

Presented by Jason A. Donenfeld

EDGESECURITY

Oct.20 (Thu) - 21 (Fri), 2016 Tokyo, Japan

CODEBLUE

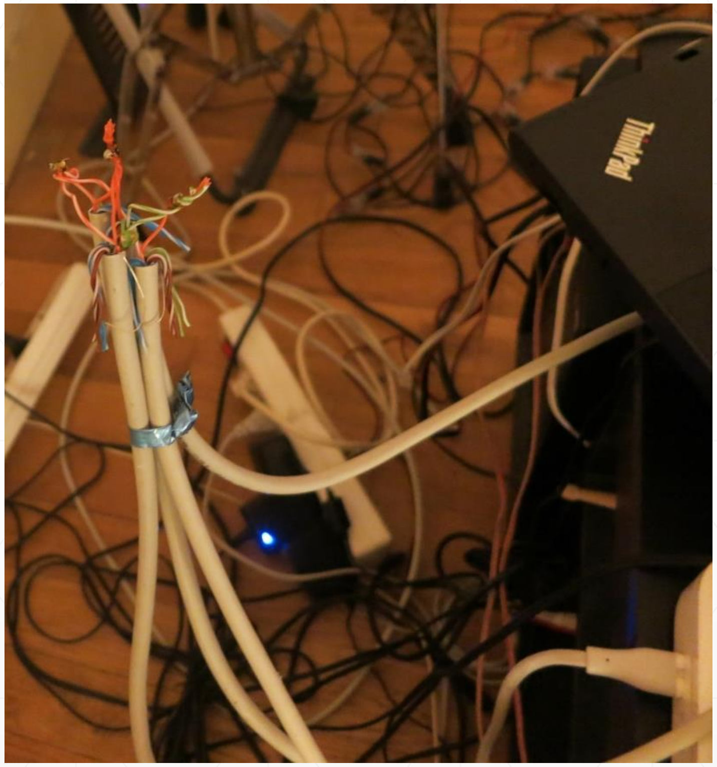
@TOKYO

Who Am I?

- Jason Donenfeld, also known as ZX2C4, founder of Edge Security (.com), a security consultancy.
- Background in exploitation, kernel vulnerabilities, crypto vulnerabilities, though quite a bit of development experience too.
- Motivated to make a VPN that avoids the problems in both crypto and implementation that I've found in numerous other projects.

What is WireGuard?

- Layer 3 secure network tunnel for IPv4 and IPv6.
  - Opinionated.
- Lives in the Linux kernel, but cross platform implementations are in the works.
- UDP-based. Punches through firewalls.
- Modern conservative cryptographic principles.
- Emphasis on simplicity and auditability.
- Authentication model similar to SSH's `authenticated_keys`.
- Replacement for OpenVPN and IPsec.

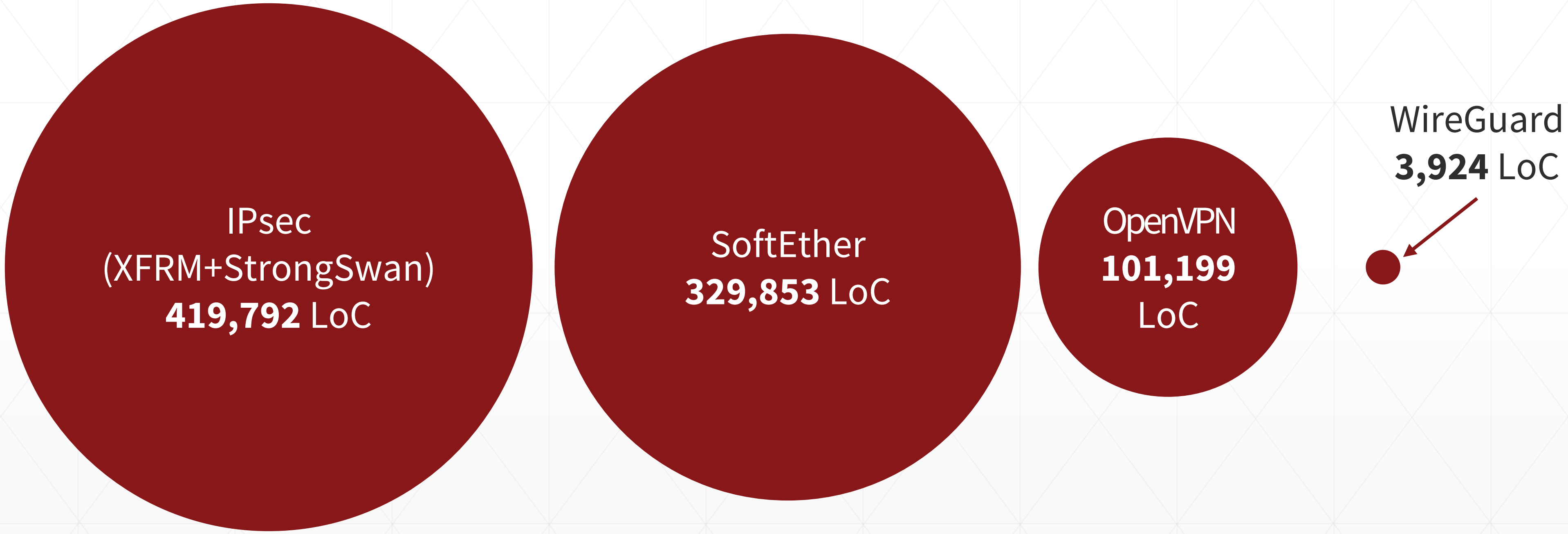


Security Design Principle 1: Easily Auditable

OpenVPN	Linux XFRM	StrongSwan	SoftEther	WireGuard
101,199 LoC Plus OpenSSL!	13,898 LoC Plus StrongSwan!	405,894 LoC Plus XFRM!	329,853 LoC	**3,924 LoC**

Less is more.

Security Design Principle 1: Easily Auditable



Security Design Principle 2: Simplicity of Interface

- WireGuard presents a normal network interface:

```
# ip link add wg0 type wireguard
# ip address add 192.168.3.2/24 dev wg0
# ip route add default via wg0
# ifconfig wg0 …
# iptables –A INPUT –i wg0 …

/etc/hosts.{allow,deny}, bind(), …
```

- Everything that ordinarily builds on top of network interfaces – like `eth0` or `wlan0` – can build on top of `wg0`.

Blasphemy!

- WireGuard is blasphemous!
- We break several layering assumptions of 90s networking technologies like IPsec.
  - IPsec involves a “transform table” for outgoing packets, which is managed by a user space daemon, which does key exchange and updates the transform table.
- With WireGuard, we start from a very basic building block – the network interface – and build up from there.
- Lacks the academically pristine layering, but through clever organization we arrive at something more coherent.

Cryptokey Routing

- **The fundamental concept of any VPN is an association between public keys of peers and the IP addresses that those peers are allowed to use.**
- A WireGuard interface has:
  - A private key
  - A listening UDP port
  - A list of peers
- A peer:
  - Is identified by its public key
  - Has a list of associated tunnel IPs
  - Optionally has an endpoint IP and port

Cryptokey Routing

Server Config

```
[Interface]
PrivateKey =
yAnz5TF+lXXJte14tji3zlMNq+hd2rYU
IgJBgB3fBmk=
ListenPort = 41414

[Peer]
PublicKey =
xTIBA5rboUvnH4htodjb6e697QjLERt1
NAB4mZqp8Dg=
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
```

Client Config

```
[Interface]
PrivateKey =
gI6EdUSYvn8ugXOt8QQD6Yc+JyiZxIhp
3GInSWRfWGE=
ListenPort = 21841

[Peer]
PublicKey =
HIgo9xNzJMWLKASShiTqIybxZ0U3wGLi
UeJ1PKf8ykw=
Endpoint = 192.95.5.69:41414
AllowedIPs = 0.0.0.0/0
```

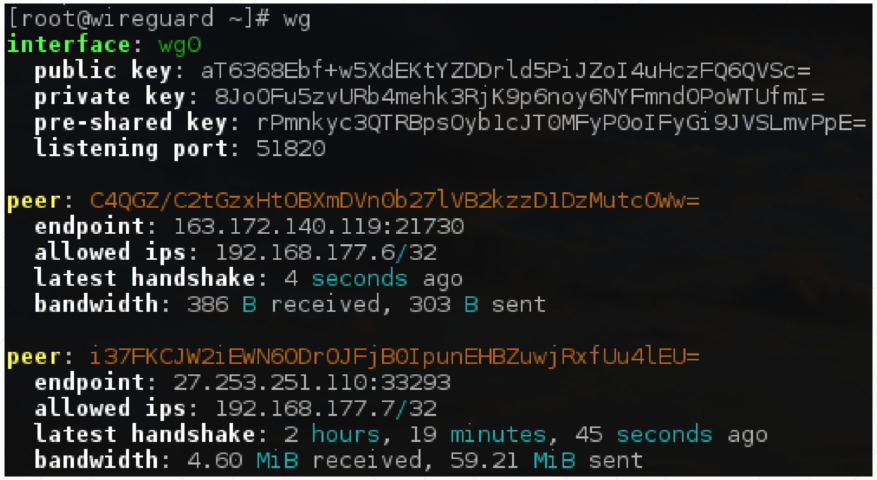
Cryptokey Routing

User space `send()`s packet. → Ordinary Linux routing table decides to give it to `wg0`. → WireGuard inspects the destination IP address of the packet to determine which peer it's for. → The packet is encrypted with that peer's session keys, and sent to the peer's endpoint.

WireGuard UDP socket `recv()`s encrypted packet. → It decrypts this packet, and in doing so learns which peer it's from. → WireGuard inspects the source IP of the decrypted packet, and sees if this actually corresponds with the peer who sent it. → If it corresponds, the packet is allowed; otherwise it's dropped.

Cryptokey Routing

-
  ```
  wg set wg0
          listen-port 2345
          private-key /path/to/private-key
          peer ABCDEF…
                  allowed-ips 192.168.88.3/32
                  endpoint 209.202.254.14:8172
          peer XYZWYAB…
                  remove
          peer 123456…
                  allowed-ips 192.168.88.4/32
                  endpoint 212.121.200.100:2456
  ```
- `wg setconf < config.file`
- `wg getconf > config.file`
- `wg show`
- `wg genkey > private.key`
- `wg pubkey < private.key > public.key`



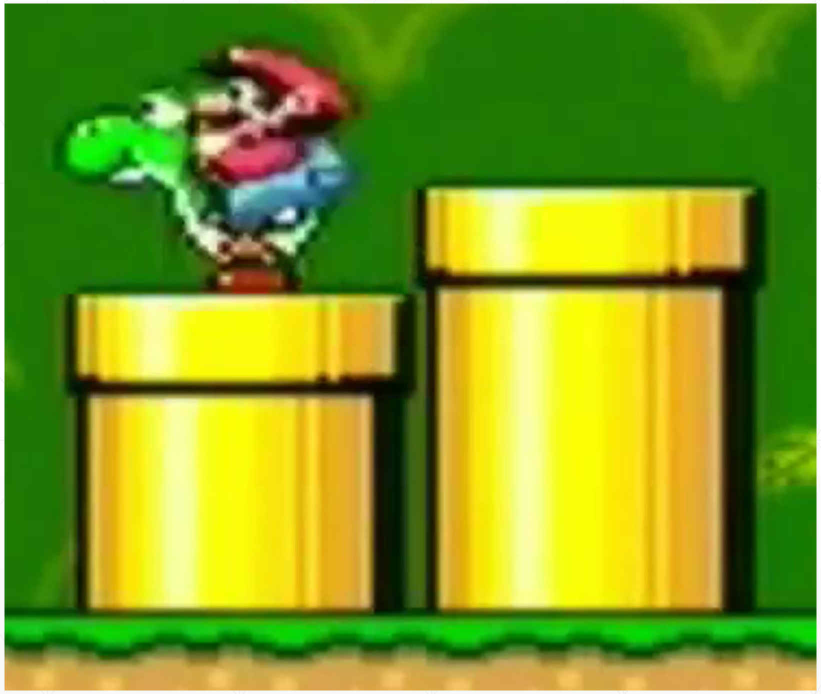
```
[root@wireguard ~]# wg
interface: wg0
  public key: aT6368Ebf+w5XdEKtYZDDrld5PiJZoI4uHczFQ6QVSc=
  private key: 8JoOFu5zvURb4mehk3RjK9p6noy6NYFmndOPoWTUfmI=
  pre-shared key: rPmnkyc3QTRBpsOyblcJTOMFyPOoIFyGi9JVSLmvPpE=
  listening port: 51820

peer: C4QGZ/C2tGzxHtOBXmDVn0b27lVB2kzzD1DzMutcOWw=
  endpoint: 163.172.140.119:21730
  allowed ips: 192.168.177.6/32
  latest handshake: 4 seconds ago
  bandwidth: 386 B received, 303 B sent

peer: i37FKCJW2iEWN6ODrOJFjB0IpunEHBZuwjRxfUu4lEU=
  endpoint: 27.253.251.110:33293
  allowed ips: 192.168.177.7/32
  latest handshake: 2 hours, 19 minutes, 45 seconds ago
  bandwidth: 4.60 MiB received, 59.21 MiB sent
```

Cryptokey Routing

- Makes system administration very simple.
- If it comes from interface `wg0` and is from Yoshi’s tunnel IP address of `192.168.5.17`, then the packet *definitely came from Yoshi*.
- The iptables rules are plain and clear.



Security Design Principle 2: Simplicity of Interface

- The interface *appears* stateless to the system administrator.
- Add an interface – `wg0`, `wg1`, `wg2`, `...` – configure its peers, and immediately packets can be sent.
- Endpoints roam, like in mosh.
- Identities are just the static public keys, just like SSH.
- Everything else, like session state, connections, and so forth, is invisible to admin.

Security Design Principle 3: Static Allocations, Guarded State, and Fixed Length Headers

- All state required for WireGuard to work is allocated during config.
- No memory is dynamically allocated in response to received packets.
  - Eliminates entire classes of vulnerabilities.
- All packet headers have fixed width fields, so no parsing is necessary.
  - Eliminates *another* entire class of vulnerabilities.
- No state is modified in response to unauthenticated packets.
  - Eliminates *yet another* entire class of vulnerabilities.

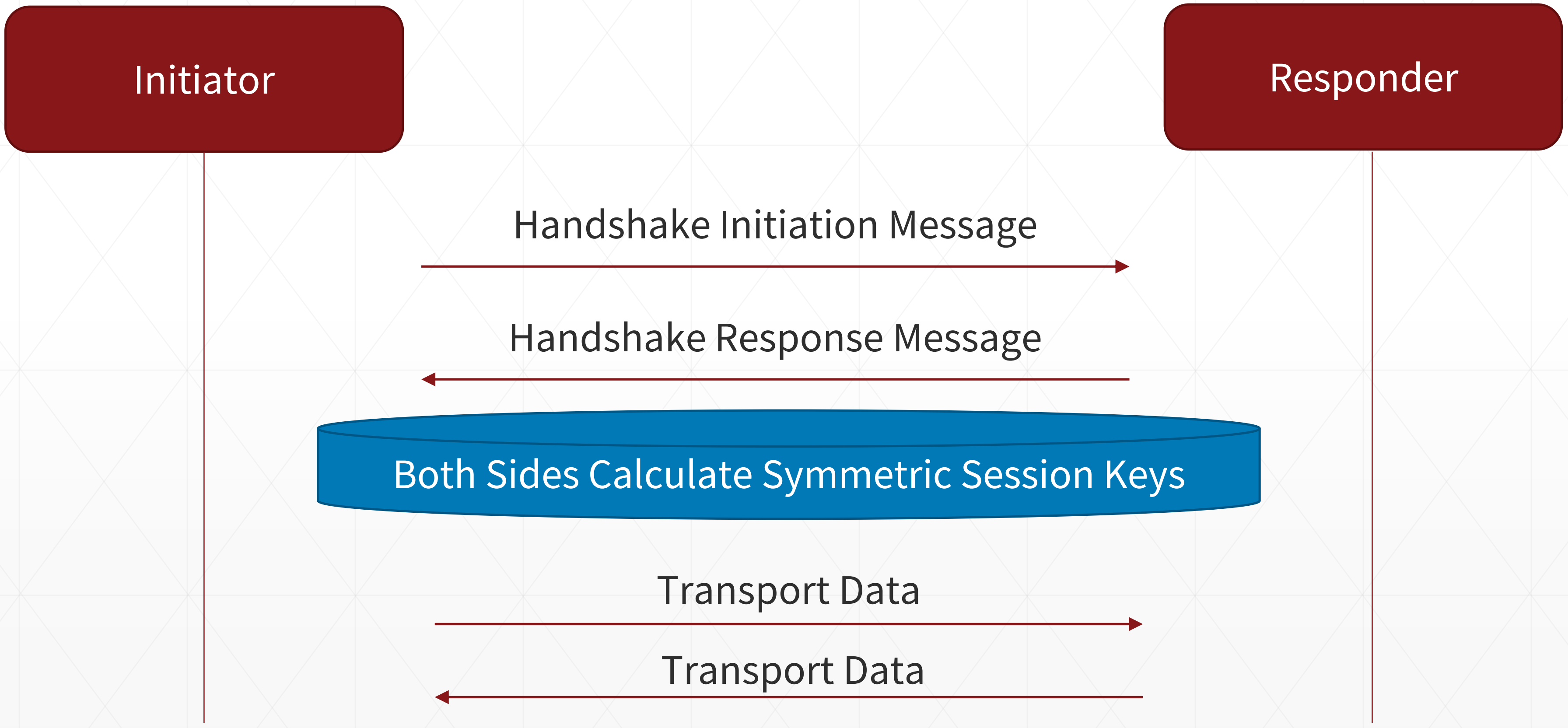
Security Design Principle 4: Stealth

- Some aspects of WireGuard grew out of an earlier kernel rootkit project.
- Should not respond to any unauthenticated packets.
- Hinder scanners and service discovery.
- Service only responds to packets with correct crypto.
- Not chatty at all.
  - When there's no data to be exchanged, both peers become silent.

Security Design Principle 5: Solid Crypto

- We make use of Trevor Perrin’s Noise Protocol Framework – noiseprotocol.org
  - Developed with much feedback from the WireGuard development.
  - Custom written very specific implementation of NoiseIK for the kernel.
- Perfect forward secrecy – new key every 2 minutes
- Avoids key compromise impersonation
- Identity hiding
- Authenticated encryption
- Replay-attack prevention, while allowing for network packet reordering
- Modern primitives: Curve25519, Blake2s, ChaCha20, Poly1305, SipHash2-4
- Lack of cipher agility!

Principles 3 + 4 + 5 → The Key Exchange


Initiator
Responder
Handshake Initiation Message
Handshake Response Message
Both Sides Calculate Symmetric Session Keys
Transport Data
Transport Data

Principles 3 + 4 + 5 → The Key Exchange

- In order for two peers to exchange data, they must first derive ephemeral symmetric crypto session keys from their static public keys.
- The key exchange is particularly well designed to keep our principles of solid crypto, static allocations, guarded state, fixed length headers, and stealthiness.
- One peer is an initiator of the exchange, and the other peer is the responder.
- Initiator and responder can swap roles at any time.
- 1-RTT
- Either side can reinitiate the handshake to derive new session keys.
- Invalid handshake messages are ignored, keeping our stealth principle.

The Key Exchange: Diffie-Hellman Review

```
private A = random()
public A = derive_public(private A)


private B = random()
public B = derive_public(private B)



DH(private A, public B) == DH(private B, public A)
```

The Key Exchange: NoiseIK

- One peer is the initiator; the other is the responder.
- Each peer has their static identity – their long term *static keypair*.
- For each new handshake, each peer generates an *ephemeral keypair*.
- The security properties we want are achieved by computing `DH()` on the combinations of two ephemeral keypairs and two static keypairs.
- Session keys = `Noise(`

```
Noise(
         DH(ephemeral, static),
         DH(static, ephemeral),
         DH(ephemeral, ephemeral),
         DH(static, static)
)
```

- The first three `DH()` make up the "triple DH", and the last one allows for authentication in the first message, for 1-RTT.

The Key Exchange: NoiseIK – Initiator → Responder

- The initiator begins by knowing the long term static public key of the responder.
- The initiator sends to the responder:
  - A cleartext ephemeral public key.
  - The initiator's public key, authenticated-encrypted using a key that is an (indirect) result of:

```
DH(initiator's ephemeral private, responder's static public) ==
DH(responder's static private, initiator's ephemeral public)
```

    - After decrypting this, the responder knows the initiator's public key.
    - Only the responder can decrypt this, because it requires control of the responder's static private key.
  - A monotonically increasing counter (usually just a timestamp in TAI64N) that is authenticated-encrypted using a key that is an (indirect) result of the above calculation as well as:

```
DH(initiator's static private, responder's static public) ==
DH(responder's static private, initiator's static public)
```

    - This counter prevents against replay DoS.
    - Authenticating it verifies the initiator controls its private key.
    - Authentication in the first message – static-static DH().

The Key Exchange: NoiseIK – Responder → Initiator

- The responder at this point has learned the initiator’s static public key from the prior first message, as well as the initiator’s ephemeral public key.
- The responder sends to the initiator:
  - A cleartext ephemeral public key.
  - An empty buffer, authenticated-encrypted using a key that is an (indirect) result of the calculations in the prior message as well as:

    ```
    DH(responder’s ephemeral private, initiator’s ephemeral public) ==
    DH(initiator’s ephemeral private, responder’s ephemeral public)
    ```

    and

    ```
    DH(responder’s ephemeral private, initiator’s static public) ==
    DH(initiator’s static private, responder’s ephemeral public)
    ```

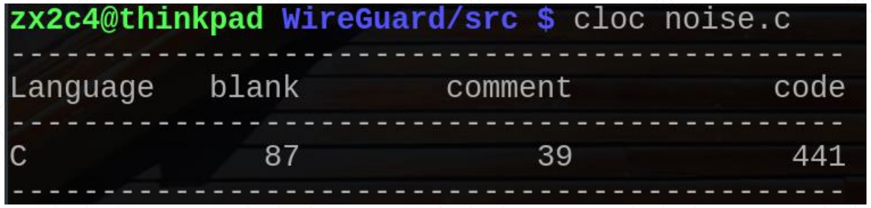
    - Authenticating it verifies the responder controls its private key.

The Key Exchange: Session Derivation

- After the previous two messages (initiator → responder and responder → initiator), both initiator and responder have something bound to these `DH()` calculations:
  - `DH(initiator's ephemeral private, responder's static public) == DH(responder's static private, initiator's ephemeral public)`
  - `DH(initiator's static private, responder's static public) == DH(responder's static private, initiator's static public)`
  - `DH(initiator's ephemeral private, responder's ephemeral public) == DH(responder's ephemeral private, initiator's ephemeral public)`
  - `DH(initiator's static private, responder's ephemeral public) == DH(responder's ephemeral private, initiator's static public)`
- From this they can derive symmetric authenticated-encryption session keys – one for sending and one for receiving.
- When the initiator sends its first data message using these session keys, the responder receives confirmation that the initiator has understood its response message, and can then send data to the initiator.

The Key Exchange

- The crypto might seem a bit complicated, but it uses a very limited set of primitives:
  - Elliptical curve Diffie-Hellman, hashing, authenticated-encryption.
- And still just 1-RTT.
- *Extremely* simple to implement in practice, and doesn't lead to the type of complicated messes we see in OpenSSL and StrongSwan.
- No certificates, X.509, or ASN.1: both sides exchange very short (32 bytes) base64-encoded public keys, just as with SSH.



```
zx2c4@thinkpad WireGuard/src $ cloc noise.c
-------------------------------------------------------------------------------
Language                     blank        comment           code
-------------------------------------------------------------------------------
C                               87             39            441
-------------------------------------------------------------------------------
```

Timers: A Stateless Interface for a Stateful Protocol

- As mentioned prior, WireGuard appears "stateless" to user space; you set up your peers, and then it *just works*.
- A series of timers manages session state internally, invisible to the user.
- Every transition of the state machine has been accounted for, so there are no undefined states or transitions.
- Event based.

Timers

User space sends packet.	• If no session has been established for 120 seconds, send handshake initiation.
No handshake response after 5 seconds.	• Resend handshake initiation.
Successful authentication of incoming packet.	• Send an encrypted empty packet after 10 seconds, if we don't have anything else to send during that time.
No successfully authenticated incoming packets after 15 seconds.	• Send handshake initiation.

Security Principle 6: Denial of Service Resistance

- Hashing and symmetric crypto is fast, but pubkey crypto is slow.
- We use Curve25519 for elliptic curve Diffie-Hellman (ECDH), which is one of the fastest curves, but still is slower than the network.
- Overwhelm a machine asking it to compute `DH()`.
  - Vulnerability in OpenVPN!
- UDP makes this difficult.
- WireGuard uses "cookies" to solve this.

Cookies: TCP-like

- Dialog:
  - Initiator: Compute this DH ( ) .
  - Responder: Your magic word is "karaage". Ask me again with the magic word.
  - Initiator: My magic word is "karaage". Compute this DH ( ) .
- Proves IP ownership, but cannot rate limit IP address without storing state.
  - Violates security design principle, no dynamic allocations!
- Always responds to message.
  - Violates security design principle, stealth!
- Magic word can be intercepted.

Cookies: DTLS-like and IKEv2-like

- Dialog:
  - Initiator: Compute this `DH()`.
  - Responder: Your magic word is “`cbdd7c…bb71d9c0`”. Ask me again with the magic word.
  - Initiator: My magic word is “`cbdd7c…bb71d9c0`”. Compute this `DH()`.
- `“cbdd7c…bb71d9c0” == MAC(key=responder_secret, initator_ip_address)`
  Where `responder_secret` changes every few minutes.
- Proves IP ownership without storing state.
- Always responds to message.
  - Violates security design principle, stealth!
- Magic word can be intercepted.
- Initiator can be DoS’d by flooding it with fake magic words.

Cookies: HIPv2-like and Bitcoin-like

- Dialog:
  - Initiator: Compute this DH ( ) .
  - Responder: Mine a Bitcoin first, then ask me!
  - Initiator: I toiled away and found a Bitcoin. Compute this DH ( ) .
- Proof of work.
- Robust for combating DoS if the puzzle is harder than DH ( ) .
- However, it means that a responder can DoS an initiator, and that initiator and responder cannot symmetrically change roles without incurring CPU overhead.
  - Imagine a server having to do proofs of work for each of its clients.

Cookies: The WireGuard Variant

- Each handshake message (initiation and response) has two macs: `mac1` and `mac2`.
- `mac1` is calculated as: `HASH(responder_public_key || handshake_message)`
  - If this mac is invalid or missing, the message will be ignored.
  - Ensures that initiator must know the identity key of the responder in order to elicit a response.
    - Ensures stealthiness – security design principle.
- If the responder is not under load (not under DoS attack), it proceeds normally.
- If the responder is under load (experiencing a DoS attack), …

Cookies: The WireGuard Variant

- If the responder is under load (experiencing a DoS attack), it replies with a cookie computed as:

```
AEAD(
   key=HASH(responder_public_key || salt),
   additional_data=handshake_message,
   MAC(key=responder_secret, initiator_ip_address)
)
```

- `mac2` is then calculated as:

```
MAC(key=cookie, handshake_message)
```

  - If it's valid, the message is processed even under load.

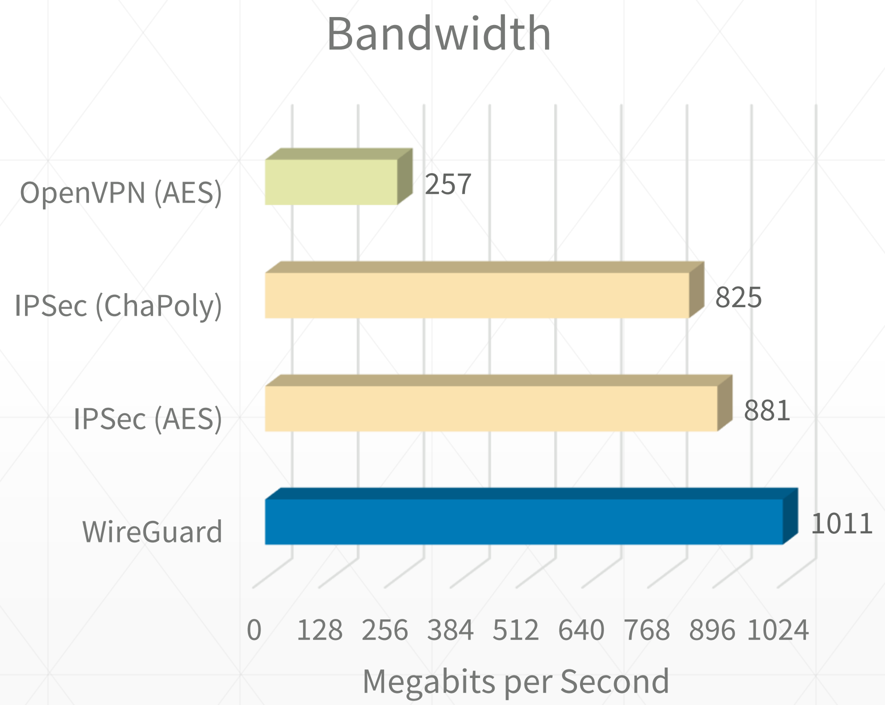
Cookies: The WireGuard Variant

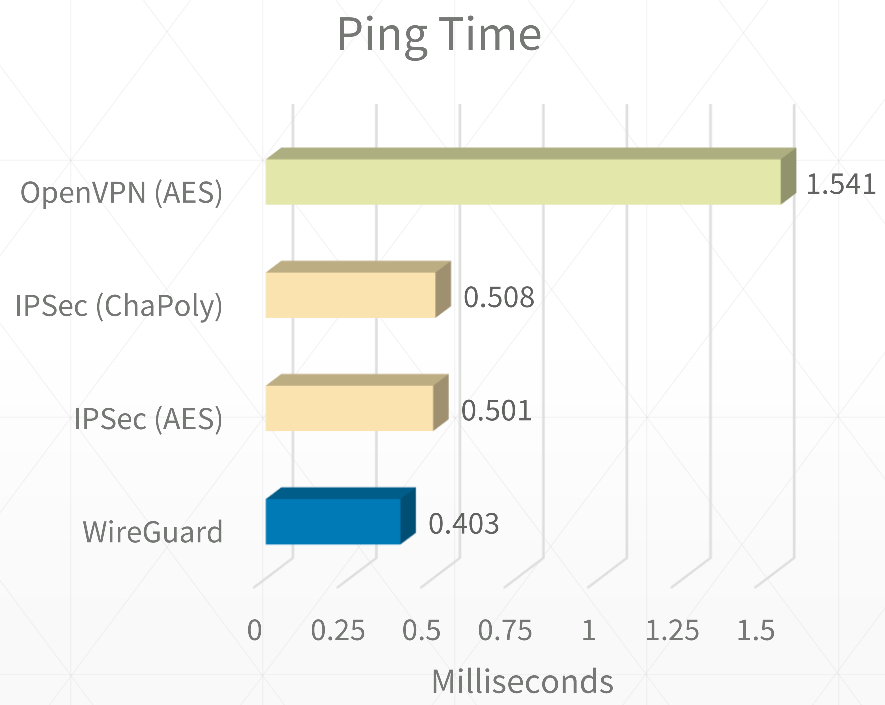
- Once IP address is attributed, ordinary token bucket rate limiting can be applied.
- Maintains stealthiness.
- Cookies cannot be intercepted by somebody who couldn’t already initiate the same exchange.
- Initiator cannot be DoS’d, since the encrypted cookie uses the original handshake message as the “additional data” parameter.
  - An attacker would have to already have a MITM position, which would make DoS achievable by other means, anyway.

Performance

- Being in kernel space means that it is *fast* and low latency.
  - No need to copy packets twice between user space and kernel space.
- ChaCha20Poly1305 is extremely fast on nearly all hardware, and safe.
  - AES-NI is fast too, obviously, but as Intel and ARM vector instructions become wider and wider, ChaCha is handedly able to compete with AES-NI, and even perform better in some cases.
  - AES is exceedingly difficult to implement performantly and safely (no cache-timing attacks) without specialized hardware.
  - ChaCha20 can be implemented efficiently on nearly all general purpose processors.
- Simple design of WireGuard means less overhead, and thus better performance.
  - Less code → Faster program? Not always, but in this case, certainly.

Performance: Measurements





WIREGUARD EDGESECURITY

Recap: WireGuard is Simple, Fast, and Secure

- **Less than 4,000 lines of code.**
  - **Easily auditable by everyone in the room.**
- Easily implemented with basic data structures.
- Design of WireGuard lends itself to coding patterns that are secure in practice.
- Minimal state kept, no dynamic allocations.
- Stealthy and minimal attack surface.
- Solid cryptographic foundation.
- Fundamental property of a secure tunnel: association between a peer and a peer's IPs.
- Extremely performant – best in class.
- Simple standard interface via an ordinary network device.
- Opinionated.

Demo

More Information

WireGuard

- Main website:
  www.wireguard.io
- Source code:
  $ git clone https://git.zx2c4.com/WireGuard
- Mailing list:
  lists.zx2c4.com/mailman/listinfo/wireguard
  wireguard@lists.zx2c4.com

Jason Donenfeld

- Personal website:
  www.zx2c4.com
- Company website:
  www.edgesecurity.com
- Email:
  Jason@zx2c4.com