Externalisation des clefs privées WireGuard sur des périphériques sécurisés

HEIG - Travail de Bachelor

Alexis Martins

05.09.2024

Sommaire

Problématique

Problématique

- Clés privées vulnérables
- Risque d’attaques sur des machines compromises
- Conséquences :
  - Déchiffrement des communications futures
  - Usurpation d’identité

↳ Sécurité insuffisante pour les environnements hautement sécurisés

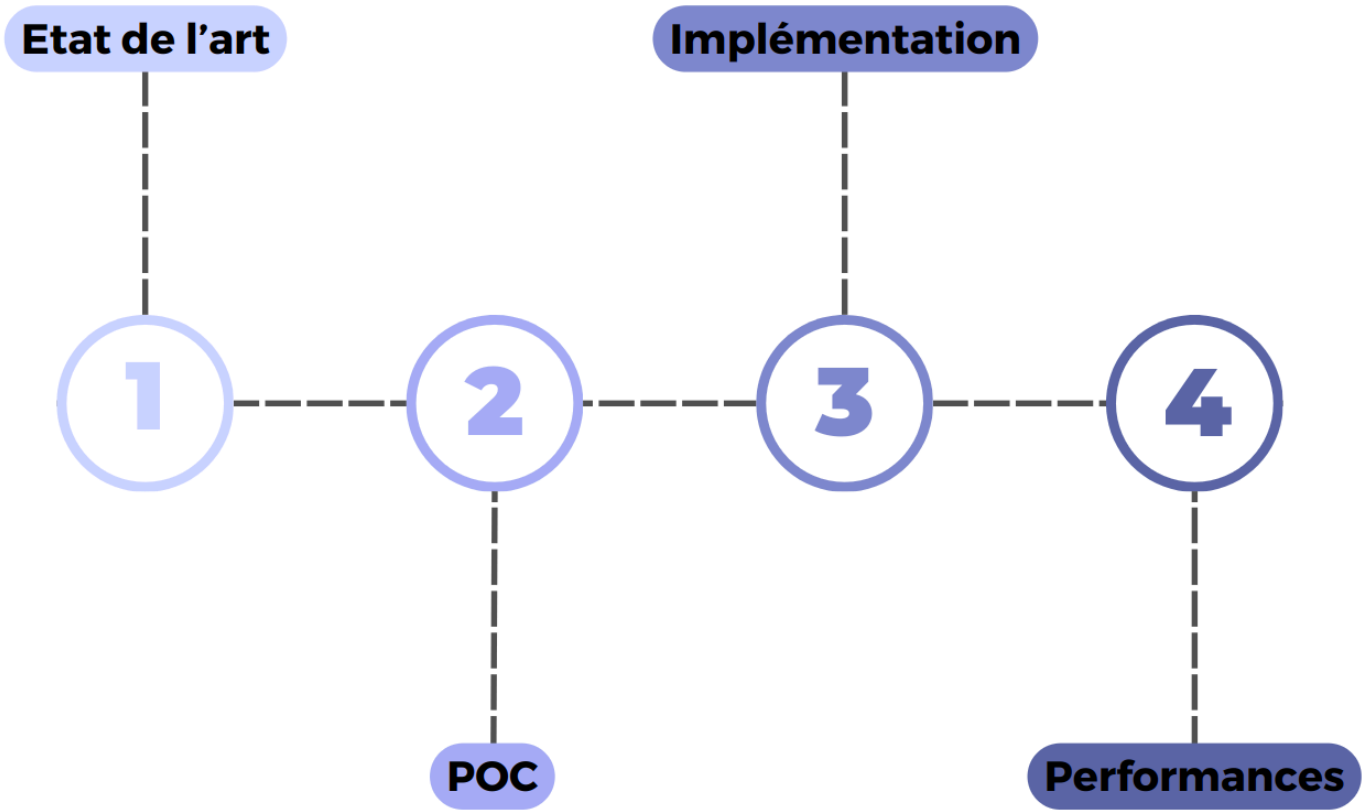
Solution

Solution

- Externaliser la clé sur un périphérque sécurisé
  - Permet de continuer de réaliser des opérations cryptographiques sans exposer la clé
  - Impossible de récupérer la clé privée

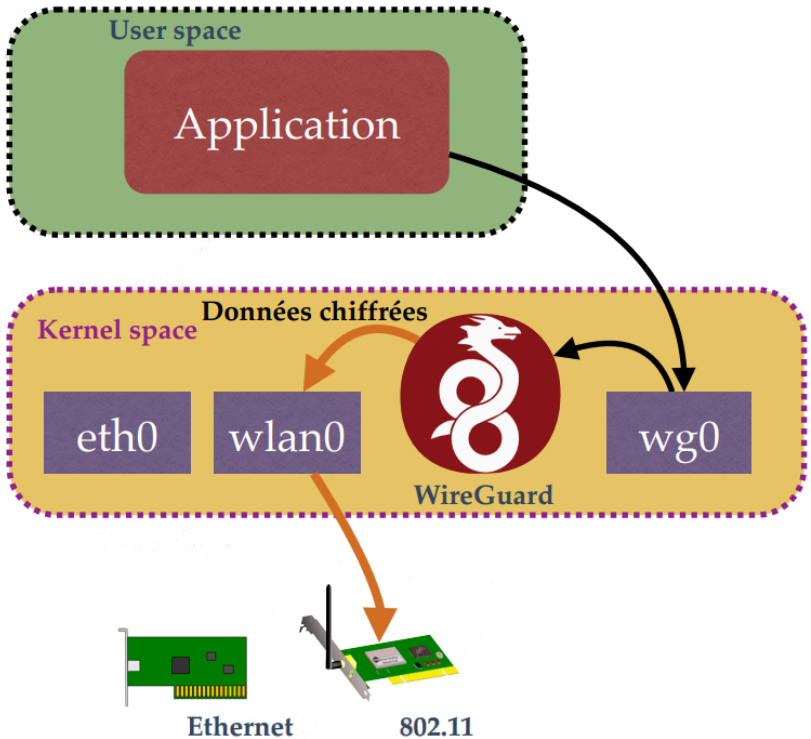
Réalisation

Déroulement du projet



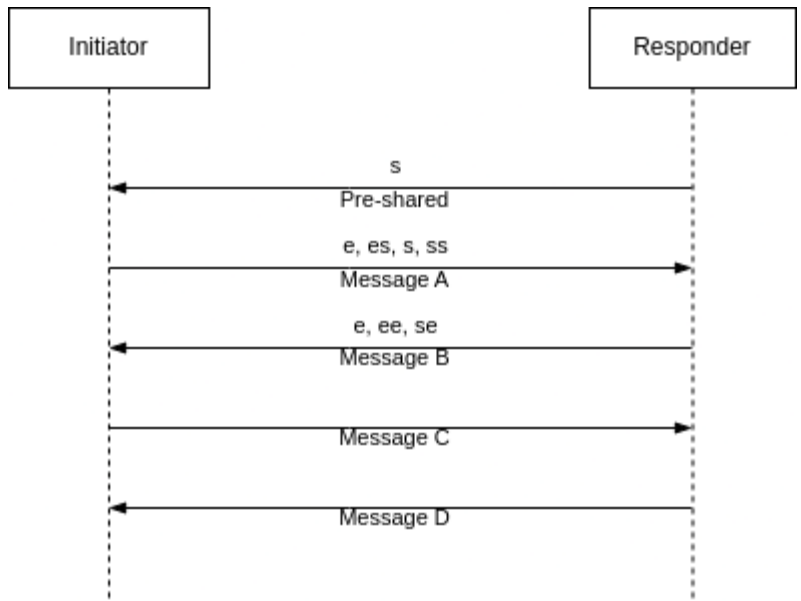
Etat de l’art

Fonctionnement de WireGuard (1)



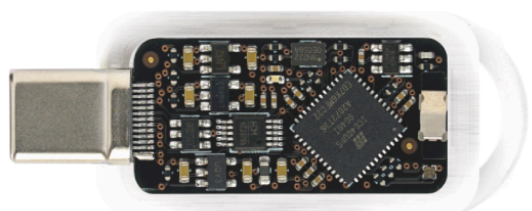
Fonctionnement de WireGuard (2)

- Utilisation de ECDH avec le Noise Protocol Framework **(NoiseIK)**



Périphériques existants

- Contraintes :
  - Doit pouvoir accueillir une clé privée de type X25519
  - Doit pouvoir réaliser une opération ECDH (déchiffrement)
  - Communication avec le langage de programmation Rust possible
- Candidats trouvés :
  - YubiKey
  - NitroKey
  - Tkey (Tillitis)



Implémentations WireGuard existantes

- Contraintes :
  - Implémentation en userspace
  - Utilisation du langage de programmation Rust
- Candidats trouvés :
  - WireGuard-rs
  - BoringTun
  - Implémentation personnalisée

Solutions existantes

- Pro Custodibus
  - Outil graphique de gestion pour WireGuard codé en Go
  - Ajoute de multiples fonctionnalités
    - Surveillance de l'utilisation réseau
    - Alertes automatiques et gestion des journaux
    - **Authentification à l'aide de périphériques externes**

POC

Technologies utilisées

- Périphérique sécurisé : NitroKey
- Protocole : OpenPGP
- Librairies Rust :
  - openpgp-card
  - card-backend-pcsc

Calcul du secret partagé

```rust
fn openpgp_case(peer_public_bytes: Vec<u8>) -> Result<[u8; 32]> {

    let backends = PcscBackend::cards(None)?;
    for b in backends.filter_map(|c| c.ok()) {

        let mut card = Card::new(b)?;
        let mut transaction = card.transaction()?;

        transaction.verify_pw1_user("123456".as_ref())?;

        match transaction.decipher(Cryptogram::ECDH(&*peer_public_bytes)) {
            Ok(res) => {
                let res_bytes: [u8; 32] = res.try_into()
                    .map_err(|_| anyhow!("Failed to convert: Vec length is not 32"))?;
                return Ok(res_bytes);
            },
            Err(e) => {
                return Ok([0; 32]);
            }
        }
    }
    Ok([0; 32])
}
```

Récupération de la clé publique

```rust
fn get_public_from_smartcard() -> Result<PublicKey> {

    let backends = PcscBackend::cards(None)?;

    for b in backends.filter_map(|c| c.ok()) {

        let mut card = Card::new(b)?;
        let mut transaction = card.transaction()?;

        if let Ok(public_key) = transaction.public_key(KeyType::Decryption) {
            let public_key_str = public_key.to_string();
            let public_key_slice = &public_key_str[public_key_str.len() - 64..];
            let mut public_key_decoded = [0; 32];
            decode_to_slice(public_key_slice, &mut public_key_decoded).expect("Failed to decode public key");
            return Ok(PublicKey::from(public_key_decoded));
        } else {
            println!("Failed to retrieve public key.");
        }
    }
    Ok(PublicKey::from([0; 32]))
}
```

Implémentation WireGuard

Implémentation WireGuard choisie

BoringTun

Points positifs	Points négatifs
Application fonctionnelle	Application complexe
Documentation disponible	En phase de restructuration
Code de bonne qualité	

Modifications apportées

- Création d'un fichier `smartcard.rs`
- Basé sur le code du POC avec quelques modifications
  - Condition permettant de prendre en charge le cas avec et sans smartcard

```
if *static_secret.as_bytes() == SMARTCARD_INDICATOR {
    ...
} else {
    ...
}
```

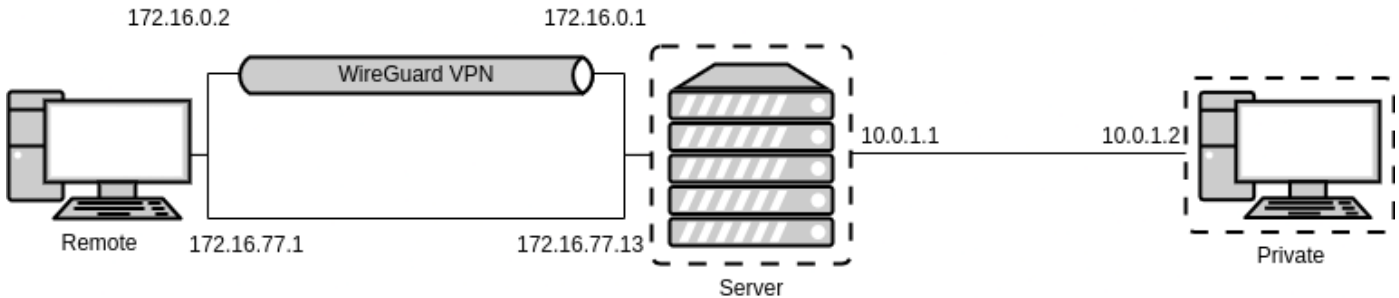
  - Ajout d'une gestion des erreurs

```
let backends_result = PcscBackend::cards(None)
          .expect("Failed to get backends");

panic!("No valid card found");
```

  - Ajout de fonctions permettant de récupérer et utiliser le PIN de la smartcard

Environnement de test



Démo


0:00
100 %
29 Aug 15:41
alexis@alexis-ThinkPad:~
alexis@alexis-ThinkPad ~
1
0:00 / 1:14

Performances

Performances

- Comprendre comment fonctionne le réseau au sein du kernel Linux
- Trouver les limitations des interfaces TUN/TAP au niveau des performances et chercher des points d'amélioration

↳ **Comprendre comment il serait possible d'améliorer la performance des VPNs en userspace**

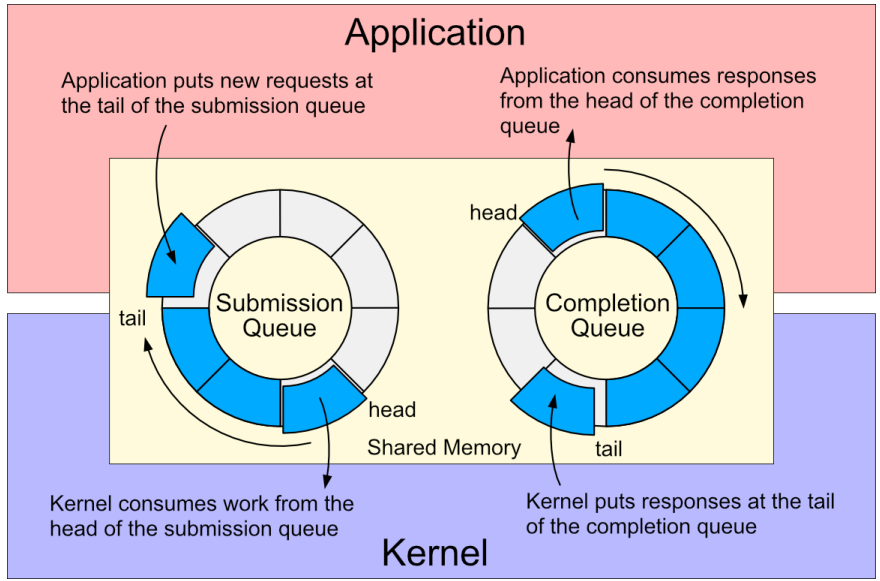
Limitations

- Changement de contexte (userspace/kernel)
- Pollution des structures du processeur
- Réduction de l'IPC

Solutions possibles

- (recv/send)mmsg
- io_uring

io_uring


Application
Application puts new requests at the tail of the submission queue
Application consumes responses from the head of the completion queue
Submission Queue
tail
head
Completion Queue
head
tail
Shared Memory
Kernel consumes work from the head of the submission queue
Kernel puts responses at the tail of the completion queue
Kernel

Programme de test

- Utilisation d'un programme réalisant des opérations réseau simples
  - Réception d'un ping sur une interface TUN
  - Création de la réponse
  - Envoie de la réponse sur l'interface TUN
- Plusieurs variantes du programme de test
  - I/O classiques
  - I/O classiques asynchrones
  - io_uring (glommio)
  - io_uring asynchrone (glommio)

Tests et résultats

Tests réalisés

- 20’000 pings
- 5 x 20’000 pings

Métriques récoltées

- Débit
- Temps de réponse moyen
- Temps total pris par les appels système
- Nombre total d’appels système

Résultats obtenus

- **Meilleur débit dans les versions classiques**, mais io_uring s’améliore avec la charge
- Temps de réponse **stable** pour la plupart des variantes
- **io_uring réduit de façon considérable le temps passé en appel système**, d’autant plus dans la version asynchrone
- Une forte charge de travail permet à **io_uring d’atteindre un état de saturation** des appels système

Conclusion

Conclusion technique

Objectif	Status
Comprendre et expliquer le fonctionnement de WireGuard	✅
Étudier les différentes possibilités d'implémentations et de matériel	✅
Réalisation de la fonctionnalité demandée	✅
Comprendre l'implémentation des sockets et de la pile réseau du noyau Linux	✅
Comprendre les limitations imposées par l'interface TUN/TAP, et les solutions choisies par les implémentations très performantes	✅
Explorer les diverses possibilités pour améliorer la performance de cette interface	✅

Conclusion personnelle

Aspects techniques	Aspects organisationnels
Gain de connaissances	Bonne gestion de mon temps
Développement de mon intérêt pour les sujets abordés	Bonne entente avec Maxime
Content du résultat produit et du déroulement du projet	

Questions ?