Université Bretagne Sud

Master 1 Ingénierie de Systèmes Complexes

Spécialité Cybersécurité des Systèmes Embarqués

Promotion 2019-2020

Étude du protocole BLE

Stage Master 1

Gidon Rémi

Avril/Juin 2020

Sommaire

Tableaux

Figures

Due aux conditions exceptionnelles imposees par la pandemie du COVID-19, j'ai réaliser un sujet fournie par le laboratoire d'informatique LAB-STICC affilié a l'université, dans le cadre du stage de premiere annee de master CSSE a l'UBS de Lorient.
Initialement prévu sur l'etude les protocoles de communication domotique (ZigBee, ZWave, Thread . . .) a l'aide d'une carte *HackRF*, j'ai du m'adapter dû au confinement et est bifurqué sur du materiel et un protocle accessible: le Bluetooth Low Energy. Sont integration dans nombre d'appareils de bureautique en ont fait un choix pour la communication avec les systemes embarques constituant les objets intelligents. l'etude du protocole est facilité par cette popularite, disposant de materiel dedie a moindre cout sur le marché ainsi ainsi qu'une floppée d'outils logiciels et d'audits de securite revelant et expliquant les vulnerabilites du protocole.

Dans le cadre du master CSSE nous étudions l'internet des objets (IoT) et leurs aspects sécurité. Le protocole réseau sans fil Bluetooth Low Energy (BLE) permet une consommation réduite pour les objets fonctionnant sur batterie, visant notamment les objets connectés. Aujourd'hui integré dans la plupart des appareils de bureautique, il est rapidement devenu populaire dans l'internet des objets.

La première itération du BLE ne répond plus aux exigences de sécurité contemporaine et même si le protocole à su évoluer depuis pour répondre à ces besoins, beaucoup d'appareils utilisent encore la version originale n'intégrant pas ces mécanismes.
Ce sont pour la plupart des appareils conçus pour fonctionner sur batterie et communiquer en point à point. On va retrouver les capteurs corporels pour santé ou fitness mais également des mécanismes plus sensibles tels des cadenas ou serrures. Les communications (incluant parfois des données personnelles) peuvent êtres interceptées, voir modifiées pour permettre des actions aux dépends de l'utilisateur.

1 Objets connectés

Avec l'explosion de l'internet de objets au cours de ces dernières années, toute une floppée d'objet du quotidien ont étés augmentés pour permettre la communication avec d'autres systèmes informatique dont nos smartphones ou encore des serveurs distants (via notre Wi-Fi). Ces objets dits intelligents étendent leur équivalent mécanique en intégrant des composants éléctroniques, permettant notamment le contrôle à distance.
Face à l'engouement du public, les constructeurs s'efforcent de proposer des objets toujours plus *intelligents* et connectés, souvent au détriment de la sécurité. Ces améliorations engendrent une augmentation de la surface d'attaque: les objets connectés sont confrontés aux mêmes challenges que ceux des systèmes informatiques traditionnels en plus de leur fonction primaire.

1.1 Architecture

Les objets connectés ont commencés par proposer des communications avec nos smartphones, notre routeur Wi-Fi ou notre ordinateur. Celles-ci permettent d'utiliser ces objets comme télécommande de contrôle (via une application dédiée la plupart du temps) ainsi que communiquer aux services distants du constructeur en s'appuyant ces fonctionnalités (Wi-Fi, données mobiles).
Ces architectures point a point connectent un appareil directement à un controleur (*smartphone*, *PC*) duquel il est dépendant pour accéder aux services distants (si il y a). On retrouve cette architecture dans les appareils autonomes qui réalisent une fonction d'augmentation sur un produit existant (comme les cadenas connectés). Ces appareils n'ont souvent pas besoin d'un service distant puisqu'ils proposent un modèle d'interactions simple et local avec un seul utilisateur.

Les objets connectés ont rapidement étés utilisés pour la mise en place des réseaux de capteurs. Cette utilisation à été largement introduite pour les particuliers avec la domotique. Les interactions avec ces réseaux ont d'abors été locales, au sein du domicile, puis globales pour permettre un controle depuis n'importe quel emplacement. Cette tendance pousse les constructeurs à exposer leurs objets connectés au réseau mondial: certains ont optés pour incorporer des puces Wi-Fi directement dans leurs objets quant a d'autres ont proposés une solution plus long terme en mettant un place une passerelle (appelée *hub*) gérant les interactions avec le monde extérieur.
L'architecture *hub* est aujourd'hui vastement utilisée dans la domotique. Un appareil dédié est considéré comme *hub* par lequel les capteurs, beaucoup plus simples, communiquent. Des protocoles spécialisés ont fait leur apparition comme *Zigbee*[1], *Z-Wave*[2], *ANT+*[3] ou encore *Thread*[4].

1.2 Protocoles

Comme évoqué précédemment, beaucoup d'objets connectés ont profité des protocoles intégrés dans les appareils utilisés comme controleur. Cela englobe le *Wi-Fi*, le *Bluetooth* et dernièrement le *NFC*. De ces trois, le *Bluetooth* à largement pris le dessus car plus adapté avec son standard *Low Energy*. Concu en tant que *WPAN* (réseau sans fil personnel) il permet de communiquer dans un rayon de 10 mètres, suffisant pour les interactions locales. Le *NFC* est assez récent, il à été conçu pour les interaction proches (une dizaine de centimètres) et intentionnelles comme les paiements. Enfin le *Wi-Fi* aurait du être le plus populaire, puisque chaque foyer dispose d'une box internet, mais sa consommation est telle qu'un objet sur batterie ne tient que quelques heures au maximum (voir les ordinateurs

portables, disposants pourtant de larges batteries). Il n'est pas adapté au besoin puisqu'il permet de hauts debits avec faible latence pour une consommation élevée la ou les objets connectés utilisent de faibles débits sur des transmission occasionnelles pour économiser leurs batteries.

Avec la démocratisation des objets connectés de nouveaux protocoles spécialisés ont fait leur apparition. Même si le *BLE* (*Bluetooth Low Energy*) s'adapte pour répondre aux besoins de ce marché, il n'a été conçu pour les objets connectés mais les objets intelligents en permettant un contrôle par l'utilisateur.
Beaucoup de grands constructeurs (notamment Google et Apple) ont developpés leur standard, le vantant et l'imposant avec leurs produits et architectures propriétaires. *Apple Home* utilise *Darwin* (iOS, macOS) comme contrôleur ainsi que son propre protocole (*HAP*) pour ses objets connectés. Google a mis en place le protocole *Thread*, interopérable avec *Google Home* (*hub*) et *Android* (*contrôleur*). Bien avant, des constructeurs specialisés ont developpés *Zigbee*, *Z-Wave* ou encore *ANT+*.
Beaucoup de protocoles se battent pour avoir accès à un marché juteux encore instable car en plein développement. Cependant beaucoup d'objets connectés n'ont pas besoin de l'interconnexion qu'apportent ces protocoles et le *BLE* est loin d'etre obsolète, étant continuellement retravaillé et proposant des améliorations intéressantes dans ce milieu.

2 Bluetooth Low energy

Le protocole a principalement été designé par Nokia pour répondre au besoin d'un protocole sans fil peu gourmand en énergie permettant la communication avec les péripheriques personnels (téléphone portable, montre, casque audio). Nommé *Wibree*, il a été intégré au standard Bluetooth sous le nom *Low Energy*.
Le Bluetooth ne comprend pas seulement un protocole mais une multitude d'entre eux (BR, EDR, HS) qui ont en commun de permettre la communication (et l'échange de données) sans fil avec des périphériques personnels. Ils font partit des protocoles *WPAN* et leur distance d'émission varie de quelques mètres jusqu'à 30 mètres.
La spécification Bluetooth 4.0, sortie en 2010, intégre le protocole LE (*Low Energy*) et permet au Bluetooth de toucher le marché des systèmes embarqués fonctionnants sur batterie.

2.1 Différences

Les autres protocoles du Bluetooth sont principalement connus et utilisés pour le transfert de contenu multimédia, que ce soit des fichiers entre ordinateurs comme de la musique avec un casque ou encore une voiture. Ils fonctionnement avec une connexion continue et un transfert en mode flux.
Le BLE, visant à reduire la consommation d'énergie, n'établie pas de connexion continue. L'appareil reste la plupart du temps en mode veille, pouvant émettre des annonces, dans l'attente d'une connexion qui aura pour effet d'arrêter la transmission d'annonces. Pour chaque requête reçu, une réponse pourra être renvoyée ou une notification mise en place périodiquement.
Les appareils BLE et Bluetooth BR/EDR ne sont pas compatibles, n'utilisant pas les mêmes technologies/protocoles et répondant à des besoin différents (voir tbl. 1).

Tableau 1: Cas d'utilisation et protocole Bluetooth adapté

Besoin	Flux données	Transmisson données	Localisation	Reseau capteurs
Appareils	ordinateur, smartphone, casque, enceinte, voiture	accessoires bureautique ou fitness, equipement medical	beacon, IPS, inventaire	automatisation, surveillance, domotique
Topologie	point à point	point à point	diffusion (1 à N)	mesh (N à N)
Technologie	Bluetooth BR/EDR	Bluetooth LE	Bluetooth LE	Bluetooth LE

2.2 Protocole

Pour permettre une interopérabilité maximale entre les appareils BLE, le standard défini 4 profils en fonction du role de l'appareil: `Peripheral`, `Central`, `Broadcaster`, `Observer`. Chaque appareil se conformant au standard ne doit implémenter qu'un seul de ces rôles à la fois.

Le *Broadcaster* ne communique qu'avec des annonces, on ne peut pas s'y connecter. Ce mode est tres populaire pour les beacons. L'*Observer* est sont opposé, il ne fait qu'écouter les annonces, n'établiera jamais de connexion.
Le *Peripheral* et le *Central* forment la seconde pair et permettent la mise en place d'une architecture client-serveur. Le *Peripheral* joue le role du serveur et est dit *esclave* du *Central* qui endosse le rôle du client et *maître*.
Le *peripheral* transmet des annonces jusqu'à recevoir une connexion d'un *central*, après quoi il arrête de s'annoncer car ne peut être connecté qu'à un *central* à la fois. Le *central* écoute les annonces de *peripherals* pour s'y connecter, puis interroge ses services via les requêtes *ATT/GATT* (*Generic Attribute*).

Couche physique

Le BLE opère dans la bande ISM 2.4GHz tout comme le Wi-Fi. Contrairement aux canaux Wi-Fi de 20MHz, le BLE découpe le spectre en 40 canaux de de 2MHz (plage de 2400 à 2480MHz).
Le protocole met en place le *saut de fréquence*, consistant à changer de canal d'émission tout les laps de temps donné, pour réduire le risque de bruit sur les fréquences utilisées (la bande ISM 2.4Ghz étant libre d'utilisation).
Sur les 40 canaux que compose le spectre, 3 sont utilisés pour la transmission d'annonce. Ils sont choisit pour ne pas interferer avec les canaux Wi-Fi car les deux protocoles sont amenés à coexister (voir fig. 1).
Les 37 autres canaux sont utilisés pour les connexions. Chaque connexion va utiliser un sous-ensemble des 37 canaux (appelé carte des canaux) pour éviter les interférences avec les autres protocoles et connexions BLE. Un seul canal transmet des données à la fois mais tous les canaux de la carte sont utilisés pour le saut de fréquences.

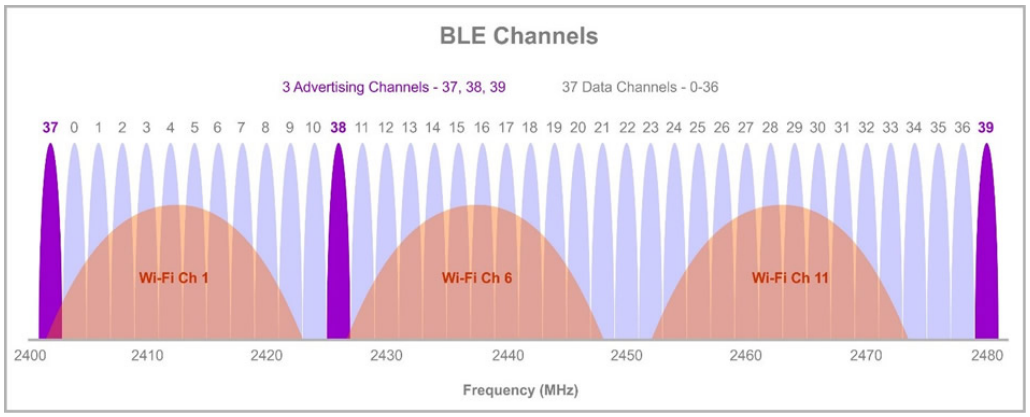

Figure 1: Répartition du spectre BLE en canaux[5]

Couche logique

1. Annonces

Le *peripheral* indique sa présence avec des annonces émises périodiquement. Ces annonces contiennent sont addresse Bluetooth (permettant une connexion) et des données qui consituent un profile (appelé *GAP*[6]). Ces données permettent aux *centrals* de savoir si il est capable de réaliser les fonctionnalités recherchées.
La spécification Bluetooth définit des profiles type pour des applications communes dans les

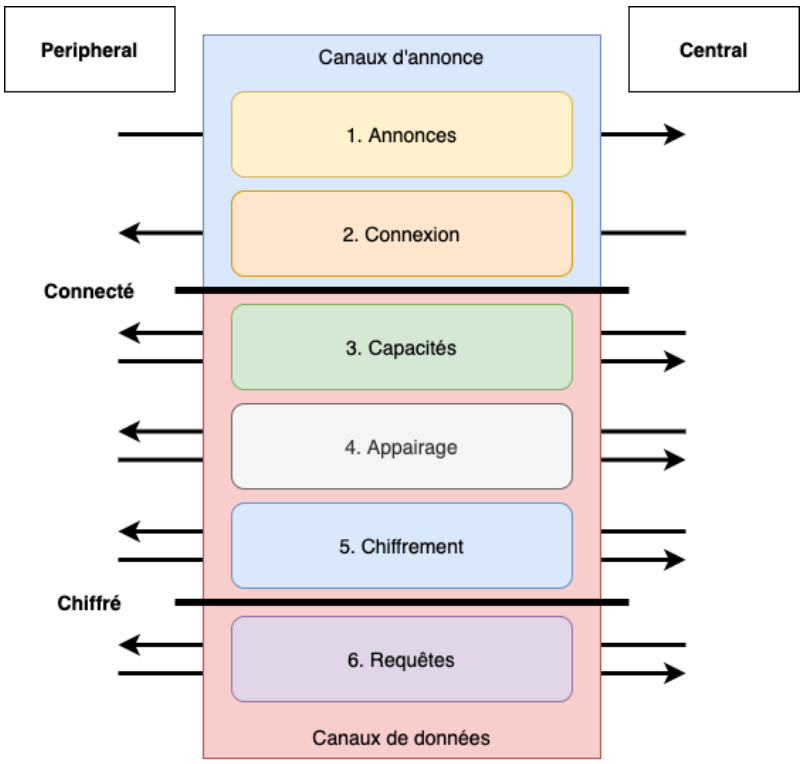

Figure 2: Étapes d'un échange BLE

appareils BLE[7]. Cela inclus par exemple les capteurs corporels pour le sport, les capteurs médicaux de surveillance (pour les diabetiques notamment), la domotique (termomètres, lampes), etc.

Dans un environnement BLE, les *centrals* ne peuvent pas reconnaître leurs *peripherals* à part avec une addresse Bluetooth fixe, mécanisme de moins en moins utilisé car vulnérable à l'usurpation. Les *peripherals* génèrent donc des adresses aléatoires et l'identification se fait via les données du *GAP* contenues dans l'annonce. Ce mécanisme permet à n'importe quel *central* de s'appairer à n'importe quel *peripheral* proposant le profil recherché.
Par exemple, une application de smartphone BLE pouvant gérer la température pourrait s'appairer et utiliser n'importe quel appareil BLE qui implémente le profil standardisé pour les termomètres dans le *GAP*.
Les profils ne sont certes pas exhaustifs mais permettent une intégration fonctionnelle avec un maximum d'appreils et prévoient un moyen d'intégrer des données propriétaires non standardisées[8].

2. Connexion

Lorsqu'un *central* reçois une annonce d'un *peripheral* auquel il souhaite se connecter, il lui envoit une intention de connexion sur les canaux d'annonce. Ce message contient tout les paramètres communs pour établir une connexion sur les canaux de données: carte des canaux utilisés, temps entre chaque saut de fréquence, nombre de canaux sautés par saut, adresse unique de la connexion (appelée *Access Adress*).
Ce message (nommé *CONNECT_REQ*) est crucial lors d'attaques car il permet la synchronisation avec une connexion pour l'écoute passive et est donc jugé sensible puisque transmit sur les canaux d'annonces avant la mise en place du chiffrement.

3. Capacités

Le BLE voulant garder une interopérabilité maximale entre les appareils mais tout les appareils ne disposant pas des mêmes fonctionnalités embarqués, il est définit plusieurs méthodes d'appairage en fonction des capacités disponibles sur les deux appareils.
Chaque appareil va transmettre ses capacités à l'autre ainsi que ses exigences sur la connexion à établir. Les capacités sont déduites des fonctionnalités présentes physiquement sur l'appareil et les exigences dépend de la version du procole actuellement supportée par celui-ci.
Les exigences comprennent la protection aux attaques *MITM* par l'authentification de l'appairage, l'établissement d'une connexion sécurisée (*LE secure connection*), la mise en place d'une session (*Bonding*) pour une reconnexion future ainsi que l'utilisation d'un canal autre que le BLE (comme le *NFC*) pour la transmission de secrets menant au chiffrement (*Out Of Band* ou *OOB*).

Tableau 2: Capacités d'entrée possibles[9]

Capacité	Description
No input	pas la capacité d'indiquer *oui* ou *non*
Yes/No	mécanisme permettant d'indiquer *oui* ou *non*
Keyboard	claver numérique avec mécanisme *oui*/*non*

Tableau 3: Capacités de sortie possible[9]

Capacité	Description
No output	pas la capacité de communiquer ou afficher un nombre
Numeric Output	peut communiquer ou afficher un nombre

Tableau 4: Capacité d'entrées/sorties de l'appareil[9]

	No output	Numeric output
No input	NoInputNoOutput	DisplayOnly
Yes/No	NoInputNoOutput	DisplayYesNo
Keyboard	KeyboardOnly	KeyboardDisplay

4. Appairage

En fonction des capacités et des exigences émises par chacun des appareils, une méthode d'appairage est sélectionnée (voir tbl. 5).

Tableau 5: Méthode d'appairage utilisée en fonction des capacités échangées[10]

	DisplayOnly	**DisplayYesNo**	**KbdOnly**	**NoIO**	**KbdDisplay**
DisplayOnly	JustWorks	JustWorks	PassKey	JustWorks	PassKey
DisplayYesNo	JustWorks	JustWorks	PassKey	JustWorks	PassKey
KbdOnly	PassKey	PassKey	PassKey	JustWorks	PassKey
NoIO	JustWorks	JustWorks	JustWorks	JustWorks	JustWorks

	DisplayOnly	**DisplayYesNo**	**KbdOnly**	**NoIO**	**KbdDisplay**
KbdDisplay	PassKey	PassKey	PassKey	JustWorks	PassKey

Je m'intéresse principalement à la methode *JustWorks*. C'est celle par défaut lorsque deux appareils ne disposent pas des capacités nécessaires pour une autre. Elle est notamment beacoup présente pour les objets connectés puisque n'intégrant pas de mécanismes pour un appairage plus complexe (claver ou écran).
Passkey permet d'authentifier l'appairage pour se protéger des usurpations d'identité (*Spoonfing* et *MITM*) puisque partageant un secret via l'utilisateur (ou un autre canal dans le cas du *OOB*). *JustWorks* ne permet pas d'authentifier les appareils et le chiffrement est moins robuste que les autres méthodes mais permet tout de même d'établir une communication chiffrée.

La méthode d'appairage choisie permet de transmettre un des matériel cryptographique: la clef temporaire (*Temporary Key*). Cette phase est plus ou moins sensible à l'écoute passive en fonction de la méthode d'appairage et des exigences émises lors de l'échange des capacités.
JustWorks avec connexion BLE 4.0 (dite *legacy*) est le mode le plus sensible puisque la clef temporaire est tout simplement zéro, ne disposant pas de moyen de transmettre une donnée par autre voie, elle peut donc etre trouvee rapidement par *brute-force*.
La connexion *LE secure*, introduite à partir de la version 4.2, utilise l'algorithme Diffie-Hellman sur courbes elliptiques (*ECCDH*) pour l'échange du matériel cryptographiques et est donc résistante à l'écoute passive (*eavesdropping*) mais toujours vulnérable à l'usurptation d'identité avec *JustWorks*.

5. Chiffrement

L'établissement du chiffrement de la connexion est ensuite réalisé par dérivation de la première clef temporaire transmise via la méthode d'appairage choisie et d'autres matériel cryptographique échangés via le protocole BLE. La clef obtenue est dite court terme (*Short Term Key*) car elle ne sera utilisée que pour cette connexion et devra être re-générée à chaque nouvelle connexion.
Il est cependant possible de mettre en place une session en stockant une clef partagée dite long term (*Long Term Key*) si cela à été exigé lors de l'échange des capacités. La clef long terme (*LTK*) est stockée et associée à l'appareil communiquant pour rétablir une connexion future sans avoir à refaire une phase d'appairage.

A partir de la comprehension actuelle du protocole BLE et du fonctionnement de l'appairage, il semble recommandé de mettre en place une connexion securisee des que possible. Il est également judicieux d'éviter la méthode *JustWorks* au maximum et stocker une session pour éviter de réitérer l'appairage.
Cependant, il est assez simple de forger un échange de capacités pour rétrograder la connexion en *legacy* et forcer *JustWorks* via les capacités échangées. C'est pourquoi certains appareils attendent des capacites et exigences minimales pour établir une connexion, sans quoi celle-ci est avortée. C'est notamment le cas d'appareils propriétaires conçus pour fonctionner ensemble.

6. Requêtes

Les échanges sont réalisés sur la base d'une architecture client-serveur. Le *central* (client) interroge le *peripheral* (serveur) avec le protocole *ATT* (*ATTribute Protocole*). Chaque requête mène soit à une réponse du serveur, soit à la mise en place d'une notification lors d'un évènemment (valeur changée ou disponible).
Les requêtes et réponses possibles sont standardisées dans le *GATT* (*Generic ATTributes*) pour permettre une interoperabilité maximale entre les appareils (comme pour le *GAP*). *GATT* et *GAP* partagent les mêmes profiles, seul la structure change. Le serveur *GATT* peut être interrogé pour établir une liste exhaustive de toutes les fonctionnalités d'un appareil la ou le *GAP* choisit ce que contient l'annonce mais est limité par la taille du paquet (31 octets).

GAP

Dans le cas des *Peripherals* et *Centrals*, le *GAP* est principalement utilisé pour établir un profil du *peripheral* permettant la décision de connexion de la part du *central*.
Pour les *Boardcasters* et *Observers* il permet la communication unidirectionnelle (*Broadcaster* vers *Observer*) via les annonces, ceux-ci utilisant la diffusion plutôt qu'une connexion point à point. On retrouve cette utilisation pour les beacons publicitaires ou de localisation intérieur.

GATT

Pour l'échange de données lors de connexion point à point, le *GATT* est utilisé en mode client-serveur. L'architecture du serveur *GATT* est en entonnoir, la plus haute couche s'appelle un *service*, il encapsule des *caractéristiques*, chacune contenant une *valeur* et un ou plusieurs *descripteurs* fournissants des informations additionnelles sur la valeur (voir fig. 3).
À chacune de ses couches (service, caracteristique, valeur, descripteur) est attribué un identifiant unique appelé *handle*. La plage des indentifiant est partagée entre toutes les couches donc si un service a l'identifiant `0x01` aucun autre service/caracteristique/attribut/descripteur ne peut l'utiliser.

Un service correspond generalement a un profil (standardisé ou non) comme par exemple un termomètre. Ce service exposerait des caracteristiques comme la température ou l'humidité. Chacune de ces caracatéristique contient la valeur (donnée brute) et des descripteurs pour indiquer l'unité ou encore un facteur ou formule pour convertir la valeur donnée en résultat exploitable.

À moins de connaître exactement l'appareil et de l'interroger à l'aveugle via les *handles* (ce qui peut être le cas entre des appareils propriétaires), il faut procéder par étape en découvrant d'abors les services disponibles, puis chaque caractéristique par service et enfin les valeurs de celles-ci.
Pour procéder à la découverte d'un appareil, le protocole *ATT* dispose d'un type de requête par couche à interroger (voir fig. 3). Une fois le service voulu trouvé (ou la cartographie totale de l'appareil realisée), on peut lire, écrire ou souscrire à des attributs directement par *handle*. Le *GATT* met en place un système de droits par attribut pour protéger la lecteur, l'écriture et la souscription par le client.

Le *GATT* définit égalemet des services standardisés appelés primaire et secondaire censés êtres présent sur tous les appareils BLE afin de connaître les fonctionnalités standardisées (service primaire) et propriétaires (service secondaire) de l'appareil. Comme les *handle* sont définies arbitrairement par le serveur *GATT*, les profils standards et leurs

services/caractéristiques sont identifiés par un *UUID* standardisé identique dans tout les appareils BLE[11].

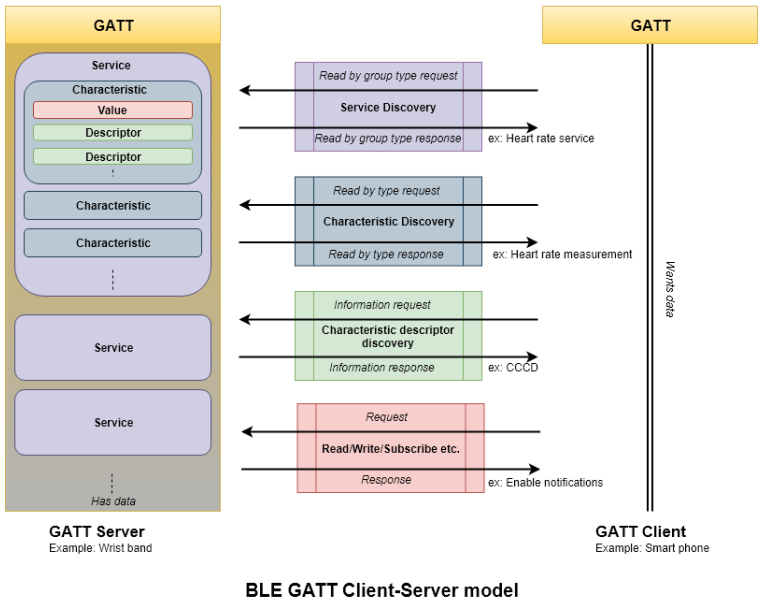

Figure 3: Client et serveur GATT[12]

2.3 Évolution

Depuis sa première itération en 2010 dans la version `4.0` des spécifications Bluetooth le BLE a évolué pour intégrer des mesures de sécurité avec l'ajout des connexions sécurisées *LE* en `4.2` puis la diversification des topologies avec l'introduction du *mesh* pour les réseaux de capteurs en `5.0` et dernièrement l'amélioration de la localisation intérieur (*Indoor Positionning System*) pour une précision de l'ordre du centimètre grâce aux systèmes angle d'arrivée et de départ (*AOA/AOD*).

3 Preuve de concept

Sujet: Étudier puis mettre en place des attaques sur le protocole *Bluetooth Low Energy* (Bluetooth Smart)

Le but est d'exposer puis abuser des failles dans le protocole BLE 4.0 (première version). Ces failles sont connues et ont pour la plupart été corrigées dans les version ultérieures du protocole (aujourd'hui en version 5.1). Néanmoins cela permet d'étudier et comprendre les mécanismes du BLE depuis sa création, puis voir les alternatives qui ont étés proposées pour mitiger ces attaques.
Je ne cible que les appareils supportant l'appairage en BLE 4.0 (dit *legacy*) avec méthode d'appairage *JustWorks*. C'est le niveau de sécurité minimal prit en charge par le protocole et très répandu dans les appareils connectés BLE. La majeure partie des appareils connectés sont simples, ne realisant qu'une fonction d'augmentation, ne disposant pas de clavier ou d'écran et ne permettent pas ainsi l'utilisation de méthode d'appairage autre que *JustWorks*.
Les mécanismes proposés à partir de la version 4.2 du BLE sont beaucoup plus robustes. Ils apportent une connexion securisée (LE Secure Connections ou LESC) se basant sur Diffie Hellmann pour l'échange de clefs ainsi qu'une nouvelle méthode d'appairage authentifiée (Comparaison numérique).

Même si le BLE à toujours proposé des mesures de sécurité, la majorité des constructeurs ne les utilisent pas et mettent en place des chiffrements au niveau de la couche application (le BLE chiffre depuis la couche lien).
Ces mesures de sécurité propriétaires sont souvent basées sur des algorithmes reconnus comme AES et des methodes telles *challenge-response* pour authentifier un appareil. Une clef unique est integrée dans chaque appareil, celle-ci sera soit distribuée au proprietaire de l'appareil lors de la création du compte ou le téléchargement de l'application associée, soit gardée par le constructeur qui transmettra directement les commandes depuis l'utilisateur à l'appareil (via l'application ou directement, si l'appareil est connecté au réseau mondial).
Ces mécanismes exposent cependant beaucoup plus d'informations que le chiffrement BLE depuis la couche lien. Les requêtes *ATT* et *GATT* transittent en clair et pour pallier à cette fuite d'informations les constructeurs évitent les requêtes standardisées dans le BLE et préfèrent utiliser des protocoles personnalisés dans la couche application, celle-ci étant chiffrée.
Ces chiffrements propriétaires sur la couche application sont hors de portée de mon sujet mais ont fait couler beaucoup d'encre. Plusieurs présentations et leurs *whitepaper* sont disponibles dans les conférences *black hat*[13], *Defcon*[14] ou encore *SSTIC*[15].

Maintenant il s'avère que beaucoup d'appareils autonomes simples ne mettent en place aucune mesure de sécurité, qu'elle soit standardisée ou propriétaire, car les données qui transittent ne sont pas jugées sensibles. C'est notamment le cas des objets domotiques autonomes comme les télécommandes pour lampes dites connectées.

3.1 Travail demandé

Mettre en place un outil basé sur un framework offensif permettant de répertorier et faciliter l'analyse des appareils et connexion BLE alentours. Cet outil est facilement portable sur diverses cartes de développement comme la Raspberry Pi car conteneurisé avec *Docker* et se basant sur du matériel USB pour l'étude du protocole.
3 ports USB suffisent pour permettre de conduire toutes les attaques proposées par le framwork offensif utilisé: 2 dongles USB BLE 4.0 et une carte BBC Micro:bit.

Le projet suppose la mise en place de 3 attaques dont une nouvelle non integrée à Mirage: - Inventaire des appareils et connexions a proximité + localisation des appareils (*scan* et *sniffing*) - Usurpation et mise en place d'un *Man In The Middle* sur un appareil selectionné (*spoofing*) - Synchronisation puis détournement par brouillage d'une connexion précédement identifiée (*hijacking*)

3.2 Architecture

Le back-end en *Python* permet le pilotage de Mirage via une API ajoutée (en rouge sur **???**). Cette API fait le lien entre le serveur HTTP/Websocket Flask (violet) et Mirage (bleu) qui pilote le matériel nécéssaire pour les attaques (vert). Côté front-end j'ai opté pour une interface web car les technologies du web (surtout du *Javascript*) ont explosé ces dernières années, rendant l'intégration d'intefaces plus simples et flexibles que les GUI applicatifs. Cette page etant une interface de controle, un framework *Javascript* (Hyperapp, en bleu clair) permet l'interactivité recherchée. Concernant les communications, il me fallait un protocole à double sens puisque l'on doit pourvoir suivre l'avancée d'une attaque en temps réel, ce que ne permet pas HTTP. Les websockets etant tres populaires dans les applications *Javascript* comme les mini-jeux, jai intégré Socket.IO (jaune) en front et back-end: c'est une librairie de communication evenementielle basee sur les websockets. Hyperapp ou Flask souscrivent aux evenements de Socket.IO pour lancer des attaques sur Mirage ou modifier le DOM (*Document Object Model*, c'est la vue HTML représentée en rouge sur **???**) en adéquation.

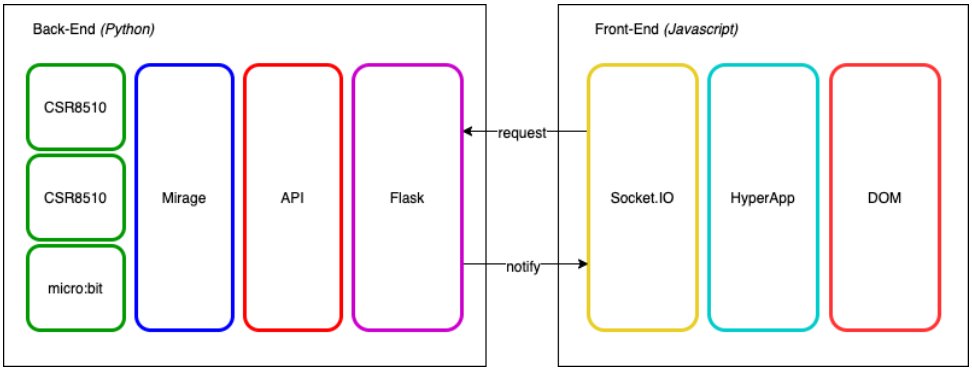

Figure 4: Architecture du système

3.3 Interface

TODO

But d'augmenter Mirage avec front-end pour demo et conduire attaques *type*

Technologie JS *Elm* (react trop lourd mais meme principe)

On retrouve la carte des appareils et connexions identifiés avec leur distance et position estimée par rapport au système (voir fig. **??**: zone rouge *Scan*).
Pour chaque cible (appareil ou connexion), des attaques sont disponibles: - Récupération du profil ou modification des transimissions par usurpation pour un appareil BLE emettant des annonces (zone bleue *Devices*). - Déconnexion des appareils ou interception des communications entre deux appareils appairés (zone bleue *Connections*).

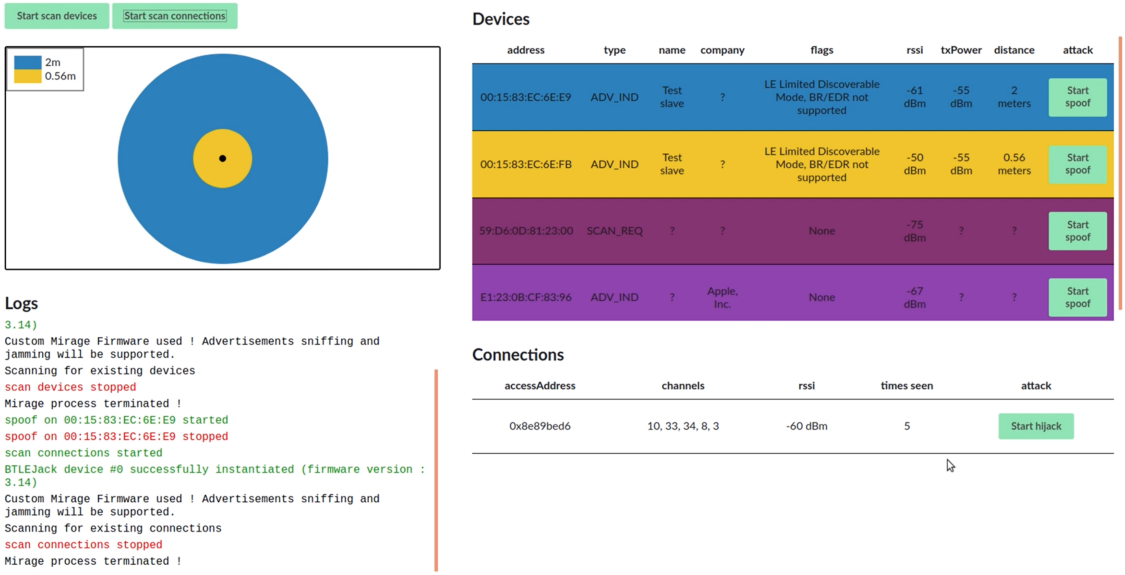

Figure 5: Interface du système

4 Étude de l'existant

4.1 Outils

Les outils offensifs sur le protocole BLE permettent de recuperer, analyser et modifier les echanges entre appareils permettant de realiser des audits de securite ou mettre en place des attaques exposant des vulnerabilites. L'analyse du traffic sans fil BLE demande une antenne couvrant la bande utilisee par les 40 canaux du protocole ainsi qu'un systeme assez rapide pour scanner puis suivre les communications lors des sauts de frequence. Les radio-logiciels (*SDR*) ne sont donc pour la plupart pas adaptés car trop lents ou trop cher pour les fonctionnalités voulues: des outils specialises dans l'analyse et l'attaque du BLE sont disponibles pour une fraction du prix.

Le premier outil, utilisé dans tout nos appareils BLE, est la puce intégrée pour les communications BLE. La recuperation (*sniffing*) et analyse ou modification d'un traffic sans fil etant interdit, ces puces utilisent un *firmware* proprietaire conforme aux roles determinés dans les specifications BLE. Même si il reste possible d'analyser le traffic (notamment en utilisant *Wireshark*[16]) entre la puce et un appareil BLE, il n'est pas possible d'etendre les capacites de celle-ci sans modifier le *firmware*.
Tout les appareils ne disposant pas d'une puce BLE dédiée, les constructeurs ont developpes des *dongles* integrant ces puces et permettant de communiquer avec tout appareil USB via une interface nommée *HCI* (*Host-Controller Interface*). Allié aux outils standard du protocole BLE comme *BlueZ*, la pile protocolaire BLE du noyau Linux, ces *dongles* permettent de découvrir les appareils a proximité et d'endosser le role de *peripheral* ou *central* pour etablir une communication avec n'importe quel autre appareil BLE. Les utilitaires *hcitool*, *hciconfig* et *gatttool* de *BlueZ* permettent par exemple de manipuler les annonces et extraire le profil *GATT* d'un appareil BLE. Meme si certains de ces *dongles* proposent des fonctionnalites interessantes comme le changement d'adresse Bluetooth (*Bluetooth Address* ou *BD*), ils n'ont pas été conçus dans une optique de sécurité, et sont peu flexibles pour un usage offensif.

La plupart des attaques sur le protocole BLE requierts un moyen d'intercepter le traffic. Les *dongles* et *puces* embarquants des firmware ne permettant pas cette fonctionnalite puisque destines au grand public, beacoups d'outils specialises ont etes developpes. On va retrouver des outils d'analyse de protocole sans fil generaux comme la *HackRF* ou sa version specialement concue pour le BLE nommée *Ubertooth One*. Ces cartes sont assez cher mais hautement personnalisables depuis les couches bas niveau. Elles demandent un certain background de connaissances sur le protocole et les modulations sans fil pour arriver a un resultat precis (comme la realisation d'une attaque).
Viennent ensuite les *sniffers* sous forme de dongle USB arrangées et plus ou moins personnalisables. Beaucoup sont basés sur les memes puces de *Nordic Semiconductor* ou *Texas Instrument* qui eux meme proposent leurs sniffers[17, 18] et logiciels[19, 20] pour l'analyse du protocole BLE. Dans les initiatives plus open-source, mais pas encore totalement personnalisable sans reprogrammation de la puce, on peut citer le Bluefruit[21] de *Adafruit*.
Enfin, un outils open-source nommé *BTLEJack* permet non seulement l'etude mais la mise en place d'une multitude d'attaques sur le protocole Ble via reprogrammation de la carte avec un firmware personnalisé. Cet outils a ete developpe pour la carte *BBC Micro:Bit*[22], une carte de developpement bon marché a but educatif, et est aujourd'hui compatible avec plusieurs autres cartes intégrant une puce `nRF51` (notamment la *Bluefruit*). Basé sur les travaux de *BTLEJack*[23] et d'autres librairies BLE en python, *Mirage*[24] permet des fonctionnalités identiques en supportant encore plus de cartes, de protocoles et d'attaques.

Il comble le manque de flexibilité des precedants outils en integrant plusieurs mecanismes permettant la mise en place d'attaques scénarisées entierement personnalisees depuis les couches protocolaires basses et facilite l'ajout de fonctionnalités au sein du framework.

4.2 Attaques

Scanning

Le *scanning* consiste à répertorier des appareils BLE à proximité. Dans le cas d'attaque on etendra l'inventaire avec les connexion établies entre 2 appareils BLE. Là ou les *dongles HCI* suffisent pour intercepter les annonces diffusées, l'analyse des communications établies requiert un *sniffer* capable de suivre les 37 canaux de données.
La pile protocolaire *BlueZ* permet le *scan* des *advertisements* (annonces) tandis que plusieurs outils precedements evoques comme *smartRF* ou *nRFSniffer* suffisent pour repérer une communication.

Spoofing

C'est l'une des étape du *Man-In-The-Middle* qui permet d'usurper un esclave BLE. Après identification de la victime (via annonce ou adresse BD), l'attaquant la clone en s'y connectant et extractant son profile *GATT*. L'attaquant peut resté connecté pour garder la victime silencieuse (un esclave connecté n'emettant pas d'annonces) puis, via un second *dongle* BLE, s'annonce comme étant l'appareil precedemment cloné.
Cette attaque est realisable en utilisant simplement un *dongle HCI* et l'utilitaire *BLueZ*. Les librairies et frameworks d'attaque discutés plus auparavant (*GATTAcker*, *BTLEJack*) integrent egalement ces mecanisme.

Sniffing

Le *sniffing* est l'analyse voir le suivis d'une connexion BLE (suivant les capacites du *sniffer* utilisé). Un premier cas de figure est l'attente d'une nouvelle connexion pour se synchroniser avec afin de suivre les echanges. La seconde option, et la plus courante, est la synchronisation avec une connexion deja etablie: la difficulté ici réside en la recuperation des parametres de connexion. Il est necessaire de retrouver la carte des canaux utilisés (*channel map*) ainsi que le *hop increment* (nombre de canaux sautés) et *hop interval* (temps entre chaque saut) pour se synchroniser, sans quoi il est impossible de suivre une connexion car les sauts impredictibles et trop frequents.
BTLEJack, et par conséquent *Mirage*, mettent en place un mecanisme permettant de retrouver ces informations de connexion a partir des echanges interceptés lors du *scanning*. Le *sniffing* de communications sans synchronisation quant a lui est une fonctionnalité très répandue et integrée a tout les sniffers BLE vu antierieurement.

Man-In-The-Middle

L'attaque *Man-In-The-Middle* concerne n'importe quelle communication: l'attaquant peut modifier les communications en vennant se placer entre l'emetteur et le recepteur ciblé, se faisant passer pour l'un apres de l'autre en usurpant leurs identités. Dans le cas du BLE on utilisera deux *dongles*, un pour usurpé l'esclave cible et un autre pour maintenir une connexion avec celui-ci. On doit d'abors usurpé l'esclave cible via du *spoofing* puis attendre la connexion d'un maitre, une fois le maitre connecté à notre *dongle* usurpateur on se retrouve en situation de *Man-In-The-Middle* entre l'esclave et le maitre: on peut

suivre et modifier le traffic avant de le retransmettre. A noté que l'on peut egalement usurpé le maitre si les appareils ciblés attendent un appareil précis pour s'appairer.
Plusieurs outils sont dediés a cette attaque car populaire et simple a mettre en oeuvre: *GATTAcker* et *BTLEJuice* facilitent la mise en place en automatisant les etapes a partir de la cible choisie. Ce sont d'assez ancien outils qui aujourd'hui souffrent de lacunes de part les technologies utilisees. Basés sur *Noble* et *Bleno*, des librairies en JavaScript basées sur NodeJS et permettant de manipuler le BLE, ils manque de flexibilité et ne permettent pas entre autre la coexistence d'appareils BLE, obligant l'utilisation de machine virtuelle pour chaque *dongle*. *Mirage* reprend le fonctionnement de ces outils, l'integrant en tant que module, mais basé sur de nouvelles librairies, notamment *PyBT*[25] permettant de simuler le comportement d'un appareil BLE en s'affranchissant des contraintes imposees par leurs equivalent JavaScript, *Noble* et *Bleno*.

Jamming

Le brouillage de communication est egalement une attaque assez populaire et implementee dans bon nombre d'outils sur le marché. Le but est de creer du bruit sur le canal au moment de la transmission pour corrompre le message, le rendant inutilisable par le recepteur. Concernant le BLE, *Ubertooth One* dispose des capacités nécessaire pour brouiller les canaux d'annonce ainsi qu'une communication etablie par retransmission simultanee. *BTLEJack* implemente egalement le brouillage au sein de son firmware personnalisé et, ayant deja un mecanisme permettant de se synchroniser avec une communication, peut egalement brouiller une communication etablie.

Hijacking

Le principe est de voler une connexion entre 2 appareils en forcant une deconnexion de l'un pour prendre sa place. Cette nouvelle attaque, implementée par *BTLEJack* et reprise dans *Mirage*, utilise les différences de *timeout* entre le *central* et le *peripheral* pour forcer le *central*, via l'utilisation de brouillage sur les paquets du *peripheral*, à se déconnecter et prendre ainsi sa place au sein de la communication.

4.3 Mirage

Après comparaison entre les outils disponibles (voir tbl. 6), j'ai choisie *Mirage* car il dispose de la flexibilite voulue pour implementee des attaques scenarisee: acces aux couches bas niveau pour recuperer informations comme force du signal et calibrage (necessaires pour calculer la position d'un appareil lors de la localisation). Il dispose egalement d'une implementation d'un *central* et *peripheral* personnalisables pour realiser un reseau de tests sur lequel verifie l'implementation des attaques. Enfin, il supporte une varieté de composants matériel[26] ainsi que toutes les attaques necessaires[27] pour la preuve de concept, rendant le developpement plus simple.

Tableau 6: Comparaison des outils pour l'étude offensive du BLE

Logiciel	*scan*	*sniff*	*mitm*	*jam*	*hijack*	*locate*	Matériel
nRFSniffer	oui	oui	non	oui	non	non	puce `nRF51`
TIsmartRF	oui	oui	non	non	non	non	puce `CC25xx`
BTLEJuice	oui	non	oui	non	non	non	*dongle HCI + Bleno/Noble*
GATTAcker	oui	non	oui	non	non	non	*dongle HCI + Bleno/Noble*

Logiciel	*scan*	*sniff*	*mitm*	*jam*	*hijack*	*locate*	Matériel
BTLEJack	oui	oui	oui	oui	oui	non	*BBC Micro:Bit*, cartes basées sur puce `nRF51`
Mirage	oui	oui	oui	oui	oui	possible	*dongle HCI*, *Ubertooth*, *nRF*, cartes compatibles avec *BTLEJack*

Mirage fournit des briques logicielles pour communiquer avec des appareils (dongles, sniffers) ainsi que decortiquer les protocoles, ce qui constitue le coeur du framework. Ces briques logicielles de base sont utilisees par des *modules* dans un but precis, par exemple realiser le brouillage d'une communication BLE. Mirage fournie un certain nombre de modules fournissant les attaques retrouvees dans les autres outils d'audit du BLE (*BTLEJack*, *GATTAcker*, ...) precedement discutes. Similaire a la philosophie d'Unix, ces modules sont specialisés dans une tache precise et peuvent etres assemblés entre eux pour realise des fonctionnalites plus complexes comme la connexion avec le module *ble_connect* puis l'extraction d'informations avec *ble_discover*. Enfin, il est possible de modifier les etapes d'une attaque via les *scénarios*: chaque module accepte un scénario surchargeant son flux d'execution et ses methodes.

Concernant le matériel necessaire, la preuve de concept necessite d'abors deux *dongles HCI* compatibles avec Mirage et supportant le changement d'adresse *BD* pour le *spoofing*. *Mirage* se base sur les numeros de constructeurs des *dongles* definit par le Bluetooth[28] pour savoir s'ils sont compatibles. Il supporte une variete des constructeur dont le *CSR8510*[29] de Qualcomm (fig. 7), puce tres populaire dans les *dongles HCI* basiques et permettant le changement d'adresse *BD*.
Concernant le *sniffer* necessaire pour la pupart des attaques, *Mirage* se basant sur *BTLEJack*, les cartes supportées par celui-ci le sont aussi par *Mirage*. Même si *Mirage* supporte d'autres cartes comme celles de developpement de *Nordic Semiconductor* ou l'*Ubertooth One*, elles ne sont pas adaptées pour mon projet car trop onereuses pour les fonctionnalités exploitees dans la preuve de concept.
Je me suis donc tourné vers les cartes compatibles avec *BTLEJack*[30]. *Bluefruit* d'Adafruit, Waveshare *BLE400*[31] et les kits *nRF51*[32] demandent une reprogrammation via un peripherique externe utilisant le port *SWD*. Ne disposant pas du materiel necessaire pour la reprogrammation, et celui-ci etant assez onéreux, j'ai choisit la *BCC Micro:Bit* (fig. 6): carte avec laquelle *BTLEJack* à ete originalement developpé.

Figure 6: Carte BBC micro:bit

Figure 7: Dongle HCI BLE CSR8510

Intégration

Mirage est un "*framework offensif pour l'audit des protocoles sans fil*"[24]. Il a ete pense pour le *pentest* (audit de sécurité) donc un usage exclusivement en ligne de commandes (*CLI*). Meme si le framework se veut beaucoup plus modulaire et extensible que ces predecesseurs (BTLEJack notamment), cette modularite a ete pensee pour l'interface en ligne de commandes.

Le fait de devoir l'integrer dans un back-end suppose une API pour communiquer avec Mirage depuis Flask. Ne disposant pas nativement de cette API je l'ai créée en m'inspirant de celle du CLI: j'ai repris la méthode d'initialisation du framework mais est remplacé les arguments en ligne de commande par une instanciation et hydratation des modules manuelle.
Le framework est fait pour fonctionner le temps d'une attaque, apres quoi il s'auto-termine, et compte sur le nettoyage par le systeme d'exploitation suite a la fin d'execution de son processus pour fermer les sockets ou vider les files (*FIFO*) utilisées dans la communication avec le matériel par lien série. Dans cette philosophie j'ai ete amené à combler ce manque car mon processus python est hôte de Mirage, il ne se ferme pas a la fin d'une attaque, donc retrouver un etat stable apres une attaque est primordial. Cela demande la suppression des caches utilisés dans l'instance de Mirage, fermeture des socket et synchronisation des fils d'executions (*threads*) supervisant le matériel pour ne pas remplir les files alors que l'attaque est terminée.

Si cela peut avoir du sense de faire une interface pour superviser une nouvelle attaque ajoutée à Mirage, reconduire le fonctionnement et interactions des attaques d'ores et deja disponibles via le CLI vers un GUI est discutable. Le *MITM* et *Hijacking* sont des attaques interactives: après usurpation d'un appareil dans le *MITM*, l'utilisateur peut modifier à la volée les paquets échangés ou communiquer via un terminal avec le *peripheral* lorsque qu'une connexion à été détournée avec du *hijacking*. Cette interaction n'est reproductible qu'en imittant un terminal sur l'interface, ce qui revient a recréer l'interpreteur deja intégré a Mirage pour n'ajouter qu'un peu de commodité à l'utilisateur (qui n'a pas à devoir utiliser le CLI depuis le conteneur Docker).

Mirage utilise des delais d'attente (`wait`, `sleep`) dans le processus principal pour attendre une certaine trame ou qu'un appareil soit dans l'etat voulu. L'interactivité est conservée par des mises a jour periodique de l'avancée de l'attaque soit via des tableaux contenant les informations trouvées soit des journaux indiquant un evenement précis. Garder cette meme interactivité dans le front-end, durant l'execution d'un scan par exemple, est plus complexe car la librairie Socket.IO du back-end ne se base non pas sur des fils d'execution mais une boucle d'evenements (principe utilisé dans NodeJS et Javascript). Les evenments lents comme les interactions avec le reseau sont toujours différées car le fil d'execution principal est lui meme interrompu par de multiples delais d'attentes. En consequence Socket.IO ne transmet jamais les evenements et le front-end reste figé. Pour remedier a cela il faut d'abors rendre Mirage compatible avec le systeme de boucle d'evenement, solution fournie par Socket.IO, qui plus est sans modifier le code du framework, par le remplacement des methodes standard de suspension du fil d'execution (`sleep`, `wait`) avec leurs equivalent en boucle d'evenement. Ensuite, plutot que de lancer l'instance de Mirage sur son propre fil d'execution, on l'insert dans la boucle d'evenement qui supervise tout le back-end. Les fils d'execution crées par Mirage suspendent ainsi son evenement dans la boucle et non le fil d'execution de la boucle d'evenement lors d'appels a `sleep` et `wait`.

5 Travail réalisé

5.1 Scan

Le scan des appareils et connexions BLE alentours se base sur un sniffer BLE, la carte BBC micro:bit dans mon cas. Les fonctionnalités de scan sont nativement supportée par Mirage et intégrées dans le firmware adéquat au framework.
C'est l'une des deux attaques retrouvé dans le front-end: l'utilisateur peut commencer un scan qui notifiera le front-end lors de découvertes, puis l'arrêter quand bon lui semble. Les appareils et connexions répertoriées ainsi que leurs informations sont disponibles sur la colonne de droite (voir fig. 8).

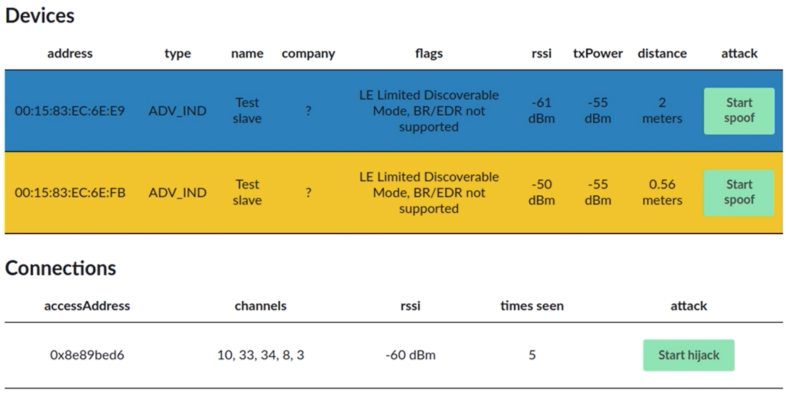

Figure 8: Appareils et connexions repertoriées à proximité

La carte BBC micro:bit n'intégrant qu'une puce nRF51, une seule commande peut être réalisée à la fois, Mirage met cependant en place du balayage de canaux basé sur un changement rapide de commandes directement dans le firmware via les minuteurs disponibles sur la carte. Ce balayage est notamment utilisé pour la découverte d'appareils BLE sur les canaux d'annonces 37, 38 et 39 ainsi que l'interception de connexions sur les 37 autres canaux de données.
Le sniffing des connexions est peu fiable dû au changement impredictible de canaux imposé par le *channel hopping*. Cette mitigation intégrée au protocole BLE rend incertain le temps pour identifié une ou plusieurs connexions BLE, la carte micro:bit changeant elle aussi de canaux pour maximiser ces chances de trouver des connexions les utilisants.

Cette attaque profite du caractère publique des canaux utilisés pour les communications, il est possible de mitiger son impact en rendant plus difficile l'identification des appareils par la réduction du nombre d'annonces émises et en choisissant le type d'annonce en fonction des besoins. Il n'est pas toujours necessaire d'emettre des annonces indirecte contenant des données du *GAP*, les annonces directes contiennent par exemple seulement le *central* recherché, rendant plus complexe la tache d'identification de l'appareil. La découverte des connexion peut egalement etre durcie en modifiant les parametres de connexion émis, plutot que d'utiliser une carte des canaux par defaut se basant sur les 37 canaux de donnees.

5.2 Localisation

Il existe plusieurs moyens de localiser des appareils, la localisation interieur est d'ailleurs un champ de recherche complexe et tres actif allant de l'inventaire d'entrepots jusqu'au profilage publicitaire.
Pour obtenir une estimation de la distance d'un appareil on peut se basé sur le temps que

met l'onde a nous parvenir (appelé *Time Of Arrival* ou *TOA*) pour en deduire la distance a partir de sa vitesse. Cependant cela requiert une information fournie par l'emetteur: l'heure d'émission, en me placant en tant qu'attaquant je ne controle pas les appareils ciblés et ne peut pas garantir la presence de cette information car peu utilisée dans les appareils particuliers.
Une autre methode beaucoup plus populaire et accessible se base sur le *RSSI* (*Received Signal Strength Indicator*). C'est un indicateur de la puissance du signal reçu en `dBm` duquel peut etre deduit la distance de l'emetteur. Cependant le BLE pouvant emettre sur une plage de puissances il est primoridial de connaitre ou trouver la puissance d'emission utilisée de la part de l'emetteur. Heureusement un standard a ete developpé pour les beacons, nommé iBeacon et intégré dans le *GAP* et le *GATT* en tant que *Tx Power* (puissance de transmission), il fourni une valeur de calibrage représentant la puissance mesurée par le constructeur a 1 metre. Meme si sa presence n'est pas garantie, le standard est tres répandu dans les appareils domestique et de bureautique.
Les entrepots et centres commerciaux utilisent sur le *fingerprinting*, c'est a dire le positionnement par rapport a des appareils proche identifiés. Chaque appareil est répertorié avec sa position et son calibrage, l'objet a localiser applique ensuite une *trilateration* a partir de la position de 3 appareils a proximité. Cette solution n'est adaptée a mon besoin car elle demande la liste des appareils identifiés, information indisponible en tant qu'attaquant.

Ensuite viens une seconde problematique, le *TOA* et *RSSI* ne fournissent pas d'information sur la direction de l'appareil, seulement une distance. Il faut alors croiser plusieurs relevés avec de la trilatération, procedé au coeur du systeme GPS visant a trouver une intersection commune entre minimum 3 cercles (voir fig. **??**), ou utiliser une matrice d'antennes pour calculer la direction du signal reçu.
Le BLE intègre depuis la version 5.1 le mecanisme d'angles de départ et d'arrivée (*AOA*/*AOD*) permettant de trouver la direction a l'aide d'une matrice d'antennes en plus d'augmenter la precision de l'ordre du metre au centimetre. Chaque antenne recois le signal avec un decalage par rapport a ses voisines, ce decalage temporel est utilisé pour approximer l'angle d'emission. En utilisant cette technique, et a partir de plusieurs emetteurs, on peut determiner une position sans se baser sur le *RSSI* ou *TOA* mais en utilisant la *triangulation*.

Pour ma part, je travail sur la version 4.0 du protocole BLE, qui n'integre pas le mecanisme *AOA*/*AOD*. Meme si il est possible de mettre en place cette methode sans la version 5.1 du BLE, cela requiert une matrice d'antennes, materiel indisponible au vu des conditions exceptionnelles.
J'ai opté pour le *RSSI* au vu de la popularité et facilité de mise en place de la methode. Les relevés sont fortement impactés par l'environnement, l'etude de celui-ci et la mise en place de modeles etant impossible dans mon cas, j'ai fixé le facteur environnmental en tant qu'espace dégagé. Je laisse tout de meme la possibilite a l'utilisateur de modifier ce facteur si besoin. Le second facteur est la sensibilité de reception, la puce nRF51 garantie une valeur à plus ou moins 6dBm avec un seuil de -30 a -90dBm, mon but est donc de reduire l'impact des ecarts de releves.

Pour les appareils en mouvement on peut ajouter de la precision avec le *dead-reckoning*, permettant de faire des previsions de position a partir de celle actuelle et des capteurs intégrés a l'appareil (gyroscope, accelerometre). On retrouve ce mecanisme pour les appareils ou applications GPS, tirant avantage des capteurs integres dans nos smartphones. Des modeles mathematiques comme le filtre de Kalman permettent egalement d'approximé les prochaines valeurs.

Le fait de se placer en tant qu'attaquant donc de ne pas controller les appareils rend le *dead-reckoning* inutilisable. J'ai choisit le modèle de pertes le plus frequement utilisé pour le *RSSI* car un modele de pertes personnalisé pour un environnement donné n'est pas envisageable puisque le projet est fait pour de la sensibilisation et est donc amené a en changer frequement.
Dans le but de reduire les ecarts, j'ai commencé par un lissage des valeurs sur une fenetre modifiable. Cela me permet de confirmer une tendance, minimisant l'impact des fluctuations du *RSSI*. Le filtre de Kalman semble etre une amelioration interessante et pourrait etre une prochaine etape.

Intégration

Dans Mirage, la localisation se base sur le travail d'identification precedement realisé par la phase de scan. Le firmware Mirage integre des informations relevees depuis la puce `nRF51` sur la transmission reçu comme la puissance du signal (*Received Signal Strength Indicator* ou *RSSI*). A partir de cette information ainsi qu'un calibrage (`TxPower`) il est possible d'approximer la distance avec le modele de pertes suivant:

$$distance = 10^{(TxPower - RSSI)/(10*n)}$$

Ou `n` est le facteur environnemental variant de 2 (espace dégagé) à 4 (zone urbaine).

Les fonctionnalités de localisation ont etes integres en tirant profit des possibilites d'extensiblité de Mirage avec l'integration d'un nouveau module: `ble_locate`. Le module partage beaucoup de fonctionnalités avec `ble_sniff`, permetttant le scan des appareils et connexions a promixité et modifie l'API de BTLEJack au sein de Mirage pour faciliter le scan de connexions. Mon but etait de faire avec ce qui etait integre au firmware Mirage car sa recompilation demande la mise en place d'un environnement precis[33]. Le module se conforme aux codes de Mirage et pourrait etre fusionné au sein du framework comme fonctionnalité supplémentaire a l'avenir.
Concernant l'interface, une carte permet de se faire une idée de la distance des appareils localisés. La carte à une échelle relative a l'appareil le plus eloigné car le BLE a une portee d'emission d'environ 10 metres, ce qui renderait indistinctibles les appareils proches si la carte couvrait toute la zone d'emission. Les autres appareils sont mis a l'echelle relativement par rapport au plus eloigné pour garder une representation realiste.

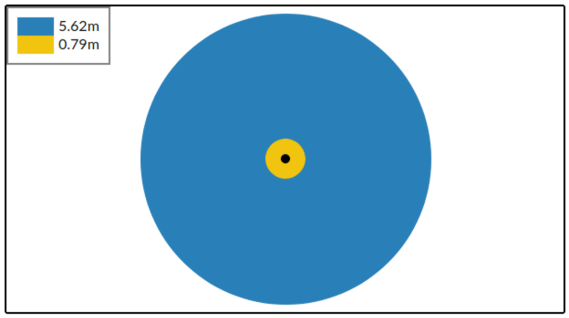

Figure 9: Carte des appareils localisés

Cette attaque concerne cependant seulement les appareils implemetant le standard iBeacon. D'une facon plus generale l'attaque est facilement mitigable en ne transmettant pas d'indication de distance dans le *GAP*, l'exposant uniquement dans le *GATT* une fois le

central identifié. Même si les beacons sont forcés d'inclure ses indications dans le *GAP*, ils ne sont pas concernés par la plupart des attaques car non connectables.

5.3 MITM

TODO

Deux dongles CSR4.0 Clonage et usurpation du *peripheral* dans l'attente d'une connexion du *central* pour voir/modifier puis relayer le traffic. Appairage sans authentification, methode de chiffrement faible (crackle pour briser le chiffrement car acces connect req) Objets connectes autonomes Module mirage ble_mitm Etre au bon endroit au bon moment, *peripheral* non connecté et *central* intention de connexion, phase appairage presente (non bonding) Utilisation de methodes d'appairage authentifiee comme PassKey ou NumComp + utilisation connexion securisee LE + mise en place de session LTK

5.4 Hijack

TODO

Matos Fct Vulns abusees Cibles Implementation Difficultes Mitigations/protections

Utilise MicroBit et CSR4.0 Prendre place d'un appareil dans une connexion. Se base sur ecoute passive puis brouillage Pas d'appairage Capteurs/actionneurs (lampes) Module mirage ble_hijack Pas acces a phase de connexion ni d'appairage => demande recover parametres de connexion Mitigé par channel hopping, protegé par appairage

5.5 Tests et validation

Dû aux conditions exceptionnelles imposees par le confinement, je n'avait pas de materiel BLE candide a disposition pour realiser mes attaques. J'ai donc mis en place un reseau de test predictible et factice entre deux CSR8510 a l'aide des modules imittant un *peripheral* (`ble_slave`) et son *central* (`ble_master`). Grace aux scenarios Mirage j'ai pu modifier leurs fonctionnement pour mettre en place un scenario de test reproductible qui m'a grandement aidé pour identifier et corriger les *bugs* lors des développements. Il semble cependant complexe d'automatiser le test de toutes les attaques implementées puisque incertaines. Les tests unitaires de code donnent un resultat attendu et identique en un temps donné, maintenant tester le réseau est beaucoup plus incertain car instable. Les attaques peuvent ne jamais se déclencher ou des interferences peuvent interrompre le deroulement.
Des tests sur le scan, la localisation, l'usurpation (*MITM*) et le détournement (*hijack*) sont tout de meme realisable manuellement via le CLI Mirage.

Le scan requiert un dongle CSR8510 emettant des annonces (voir plusieurs) et minimum une connexion etablie. Le scenario `MockSlave` (modiciation du module `ble_slave` a des fins de test) s'annonce jusqu'a ce qu'une connexion soit faite, permettant de tester le scan d'appareils alentours. Quant aux connexions etablies cela requiert une pair de `MockSlave` et `MockMaster` avec leurs CSR8510 respectifs: les scenarios de test emettent des requetes preiodiquement une fois connecté pour generer un traffic.
Comme évoqué precedemment, le scan des connexions etablie est incertain et peut durer indéfiniement. On pourrait mitiger ce probleme en parrallelisation plusieurs micro:bit, chacune scannant une partie des 37 canaux de données. Dans l'ideal il faudrait 37 micro:bit pour les canaux de données et 3 pour ceux d'annonces. Mirage dispose d'ailleurs du balayage pour pallier a ce probleme de parralelisation lors du scan d'appareils.

Les `MockSlave` sont programmés pour emettre `TxPower` dans leur annonce pour permettre la localisation, calibrage precedemment relevé a 1 metre du CSR8510 puis intégré au *GAP*. On ainsi peut mesurer l'acuité des distances en les comparants a la realité. Apres avoir fait plusieurs tests a des distance variant de moins d'un metre a 10 metres, le RSSI est fortement affecté par la distance et les objets entre l'emetteur et le recepteur. A une distance de moins d'un metre on trouve un resultat avec une precision de l'ordre de 30 centimetre et si l'on se place hors de la ligne de vue de l'emetteur, la distance calculée augmente car le RSSI diminue dû aux pertes. Un autre probleme est le seuil de sensibilité annoncé de -30 a -90dBm par le constructeur de la puce nRF51 mais d'apres mes test (basés sur le firmware Mirage) celui-ci varie entre -45dBm au plus proche jusqu'a -70dBm a la distance maximale hors de la ligne de vue (apres quoi le signal est considéré perdu). La precision des resultats aux bornes de ces valeurs est faussée car un appareil se trouvant a 10 comme a 45 centimetres aura un RSSI de -45dBm, idem pour un appareil proche de la distance maximale de reception.
Une amélioration intéressante serait l'ajout de points de releves pour permettre la triangulation a partir d'un seul recepteur. Il suffierait des lors de se deplacer entre chaque releve pour avoir une idée des appareils localisés dans l'espace, et non seulement une distance. Bien sur il est possible d'ajouter 2 autres RaspberryPi avec leurs micro:bit pour permettre une triangulation a partir d'un seul releve, en admettant qu'elles soient disposees correctement.

L'usurpation demande 2 CSR8510 pour mener l'attaque et 2 pour la testée ainsi qu'un ordre précis dans l'execution. Un dongle CSR8510 ecoute les annonces dans l'attente du `MockSlave`. Une fois trouvé, il s'y connecte, le clone et maintient la connexion pour stopper l'emission d'annonces. Un second CSR8510 usurpe le `MockSlave` precedemment cloné et s'annonce en tant que tel dans l'attente du `MockMaster`. Des lors que le `MockMaster` se connecte a l'usurpateur, un scenario modifie les paquets échangés entre `MockSlave` et `MockMaster`, rendant vérifiable le fonctionnement de l'attaque depuis leurs CLI respectifs.

Pour conclure, le détournement de connexion ... hijack TODO

6 Conclusion

TODO Plus projet whitepaper qu'entreprise, bcp porte sur R/D et decouverte/comprehnsion de sujets.
Interessant pour rapport et background networks/BLE
Dommage conditions par appareils BLE pour test car plus interessant et factuel qu'une connexion factice

[1]. https://zigbeealliance.org/.

[2]. https://www.z-wave.com/.

[3]. https://www.thisisant.com/.

[4]. https://www.threadgroup.org/.

[5]. https://www.accton.com/Technology-Brief/ble-beacons-and-location-based-services/.

[6]. https://www.bluetooth.com/specifications/assigned-numbers/generic-access-profile/.

[7]. https://www.bluetooth.com/specifications/gatt/services/.

[8]. https://www.silabs.com/community/wireless/bluetooth/knowledge-base.entry.html/2017/02/10/bluetooth_advertisin-hGsf.

[9]. https://www.bluetooth.com/blog/bluetooth-pairing-part-1-pairing-feature-exchange/.

[10]. https://www.bluetooth.com/blog/bluetooth-pairing-part-2-key-generation-methods/.

[11]. https://www.bluetooth.com/specifications/gatt/services/.

[12]. https://fr.mathworks.com/help/comm/examples/modeling-of-ble-devices-with-heart-rate-profile.html.

[13]. https://www.blackhat.com/.

[14]. https://www.defcon.org/.

[15]. https://www.sstic.org/.

[16]. https://www.wireshark.org/.

[17]. https://www.nordicsemi.com/Software-and-tools/Development-Kits/nRF51-Dongle.

[18]. http://www.ti.com/tool/CC2540EMK-USB.

[19]. https://www.nordicsemi.com/Software-and-tools/Development-Tools/nRF-Sniffer-for-Bluetooth-LE.

[20]. http://www.ti.com/tool/PACKET-SNIFFER.

[21]. https://www.adafruit.com/product/2269.

[22]. https://microbit.org/.

[23]. https://github.com/virtualabs/btlejack.

[24]. https://homepages.laas.fr/rcayre/mirage-documentation/index.html.

[25]. https://github.com/mikeryan/PyBT.

[26]. https://homepages.laas.fr/rcayre/mirage-documentation/devices.html.

[27]. https://homepages.laas.fr/rcayre/mirage-documentation/modules.html.

[28]. https://www.bluetooth.com/specifications/assigned-numbers/company-identifiers/.

[29]. https://www.qualcomm.com/products/csr8510.

[30]. https://homepages.laas.fr/rcayre/mirage-documentation/devices.html#btlejack-device.

[31]. https://www.waveshare.com/ble400.htm.

[32]. https://www.waveshare.com/nrf51822-eval-kit.htm.

[33]. http://docs.yottabuild.org/.

[34]. Cayre, R.; Roux, J.; Alata, E.; Nicomette, V. and Auriol, G.: Mirage : Un framework offensif pour l'audit du bluetooth low energy (2019), https://hal.archives-ouvertes.fr/hal-02268774.

[35]. SIG, B.: Bluetooth spécifications version 4.2 (2014), https://www.bluetooth.com/specifications/archived-specifications/.

[36]. Jasek, S.: Gattacking bluetooth smart devices, http://gattack.io/whitepaper.pdf.

[37]. Ryan, M.: Bluetooth: With low energy comes low security, https://www.usenix.org/system/files/conference/woot13/woot13-ryan.pdf.