300 secondes chrono : prise de contrôle d'un infodivertissement automobile à distance

Philippe Trébuchet et Guillaume Bouffard
philippe.trebuchet@ssi.gouv.fr
guillaume.bouffard@ssi.gouv.fr

Laboratoire Architectures Matérielles et Logicielles (LAM)
Agence Nationale de la Sécurité des Systèmes d'Information (ANSSI)

Résumé. Les véhicules connectés intègrent de nombreuses technologies de communications sans-fil à distance, comme celles exploitant les protocoles Bluetooth ou WiFi. Si le gain en confort d'utilisation et d'interaction est notable, la mise à disposition de ce type d'interfaces augmente les risques en matière de cybersécurité.

Dans cet article, nous analysons l'implémentation de la pile Bluetooth embarquée dans le système d'infodivertissement d'un véhicule du début des années 2020. En particulier, et malgré la mise en place de mesures de sécurité, de défense en profondeur et des mises à jour, une vulnérabilité a pu être exploitée. Elle permet à un attaquant distant, sans authentification et sans interaction avec les passagers, de prendre le contrôle du système d'infodivertissement en exécutant un code arbitraire. Ce code peut, entre autres, générer des commandes CAN à la volée ayant une influence directe sur le comportement et la sécurité du véhicule ciblé. Les répercussions potentielles sont donc extrêmement sérieuses.

Cette vulnérabilité, corrigée après notification, souligne l'importance d'une vigilance continue face aux risques de sécurité dans les véhicules connectés.

1 Introduction

Les véhicules modernes, dits *connectés*, sont devenus de véritables ordinateurs sur roues, intégrant, au travers de leur système d'infodivertissement, de nombreuses technologies de connectivité sans-fil comme le Bluetooth, le WiFi ou encore les réseaux cellulaires. Toutes ces interfaces de communication offrent de nombreux bénéfices en termes de confort et de fonctionnalités pour le conducteur et ses passagers. Cependant, l'augmentation de l'interconnectivité s'accompagne également de nouveaux risques en matière de cybersécurité.

En effet, chaque interface de communication embarquée au véhicule représente une nouvelle surface d'attaque potentielle pour des acteurs malveillants. La pile Bluetooth, par exemple, est devenue un élément clé des

systèmes d'infodivertissement et de télématique automobile. Cependant, les implémentations de cette technologie dans les véhicules peuvent parfois présenter des vulnérabilités [12] qui pourraient être exploitées à distance.

La sécurité des véhicules connectés est ainsi devenue un enjeu majeur [17]. Une compromission de ces systèmes pourrait avoir des conséquences graves, allant d'une atteinte à la vie privée du conducteur (géolocalisation furtive, écoute de conversations dans l'habitacle etc.) jusqu'à des risques pour la sécurité physique des occupants et des personnes à l'extérieur. Il est donc essentiel d'anticiper et d'identifier les risques de sécurité dans l'implémentation du système d'information des véhicules.

Les différentes technologies embarquées présentent des niveaux de risque de sécurité différents. Le Wi-Fi, par exemple, nécessite généralement une authentification par mot de passe et une connexion active pour être accessible, limitant ainsi les vecteurs d'attaque potentiels. À l'inverse, le Bluetooth offre une surface d'exposition significativement plus large : il reste actif indépendamment de toute connexion et demeure accessible même en l'absence d'utilisateur connecté. Cette caractéristique intrinsèque fait du Bluetooth un protocole particulièrement sensible en termes de sécurité, où chaque connexion potentielle peut représenter une menace.

Dans cet article, nous nous intéressons à l'analyse de la sécurité de la pile Bluetooth. Elle est embarquée dans l'environnement d'infodivertissement du véhicule et est déployée dans différents modèles, de différents constructeurs, de véhicules connectés. En analysant la sécurité de cette implémentation, nous avons découvert un risque de sécurité permettant à un attaquant de prendre le contrôle de l'infodivertissement, à distance et sans interaction avec l'utilisateur.

Cet article est organisé comme suit : la section 2 offre un aperçu de la sécurité des véhicules connectés, avec un focus particulier sur notre cible et le modèle d'attaquant retenu. Dans cet article, nous nous focaliserons sur la sécurité de l'implémentation de la pile Bluetooth. La section 3 rappelle le fonctionnement de cette pile et résume les informations publiques liées à l'implémentation embarquée dans notre cible. La section 4 détaille l'analyse de sécurité que nous avons réalisée, les vulnérabilités découvertes, ainsi que l'exploitation associée. La section 5 évalue l'impact des travaux présentés. Enfin, la section 6 offre un état de l'art des travaux connexes à cette étude avant de conclure cet article et de proposer quelques perspectives.

2 La sécurité des véhicules connectés

La sécurité des véhicules connectées est devenue, au fil des ans, un enjeu majeur des constructeurs. Les préjudices que peuvent engendrer des attaques sont très importants [25]. Les données manipulées par ces véhicules sont de plus en plus importantes en volume et comportent de plus en plus de données personnelles. Ainsi, les véhicules sont devenus des cibles de choix pour les groupes offensifs. Les objectifs de ces attaquants sont très variées, allant du simple vol [35] à la prise de contrôle de véhicules autonomes [10, 12] (pouvant provoquer, par ce biais, des dommages physiques aux occupants ou aux autres usagers), en passant par l'espionnage de l'habitacle [23]. Ces exemples s'étalent sur une période de temps allant de 2013 à nos jours et témoignent d'une sophistication croissante des attaques comme de la diversité des cibles affectée au sein du parc automobile mondial.

Dans ce travail, nous avons étudié un véhicule représentatif du début des années 2020, dans une finition haut de gamme. Ce type de véhicule est encore en circulation.

2.1 Description de la cible

Parmi l'ensemble des unités de commande électroniques (ECUs) du véhicule, celui exposant le plus d'interfaces externes est l'infodivertissement. Cet ECU est en effet connecté physiquement sur les différents bus de communication du véhicule et est en liaison directe avec le calculateur télématique (TCU). Ce calculateur étant central dans le fonctionnement du véhicule, nous nous sommes concentrés sur lui dans le cadre de l'étude présentée dans cet article.

Sur notre véhicule cible, cet ECU embarque une version 4.4 d'Android, doté d'un noyau 3.0.35. Au début de l'étude, ce type d'ECU équipait la plupart des véhicules neufs du constructeur cible.

La plateforme matérielle embarque un i.MX 6 DualLite, 2 Go de RAM et une eMMC de 32 Go contenant à la fois le système et les données de navigation.

L'analyse du *firmware* embarqué montre en outre une certaine maturité au niveau des fonctions de sécurité embarquées activées sur la cible de l'étude :

— *High Assurance Boot* (HAB). Il s'agit d'une fonctionnalité des microprocesseurs i.MX 6 qui permet de n'exécuter que du code signé sur la plateforme. Ce qui empêche toute modification du *firmware*.

— randomisation de l'espace d'adressage (ASLR), c'est-à-dire que les adresses mémoires auxquelles sont placés les différents composants d'un processus sont non-predictibles.
— *Data Execution Prevention* (DEP), qui recouvre plusieurs réalités : les piles sont non-executables, le W^X est activé, de même que les analogues des fonctionnalités micro-architecturales *Supervisor Mode Execution Protection* (SMEP) et *Supervisor Mode Access Prevention* (SMAP).
— Les symboles de `DEBUG` ont été retirés du noyau et le fichier `config` de celui-ci n'apparaît nulle part dans l'image disque.

Bien sûr, en 2025, on pourrait argumenter qu'il manque nombre de fonctionnalités à cet inventaire, car les versions ultérieures d'Android ont beaucoup d'autres contre mesures. Toutefois, compte tenu de la date de fabrication de la plateforme matérielle et de la version d'Android présente dans l'équipement on peut estimer que les mesures de sécurité disponibles ont été activées. On peut mettre en perspective ces propos en comparant notament avec ce qui est mentionné dans [7] où les analystes constatent que l'ASLR est désactivé sur leur plateforme pourtant plus récentes de 4 ans.

Une application nommée `Settings` est présente sur la cible de l'étude. Cette application ne semble pas accessible via l'utilisateur par défaut. L'analyse de cette application montre qu'il y est fait mention de paramètres en lien avec le WiFi, le Bluetooth, le SDIO,[1] et l'USB. L'examen minutieux du véhicule utilisé pour l'étude ne montre aucun connecteur SD, mais qu'un connecteur USB est présent.

Concernant la surface d'attaque de cet ECU, il embarque une connectivité Bluetooth et WiFi. La version du noyau laisse supposer qu'il embarque bon nombre de CVE non patchées. En particulier, au niveau des *drivers* de ses interfaces externes (Bluetooth, USB et WiFi). En outre l'ancienneté des composants laisse aussi à penser qu'ils ont aussi leur lot de vulnérabilités connues.

L'USB n'est pas accessible en dehors du véhicule, et le WiFi n'est pas activé de base sur la cible de notre étude. Le Bluetooth est donc l'interface apparaissant comme la plus naturelle à étudier. A ce niveau, le noyau 3.0.35 est vulnérable à la faille Blueborne [3] (`CVE-2017-0781` et `CVE-2017-0782`). Mais les tests de présence de cette vulnérabilité s'avèrent négatifs. Cet état de fait s'explique par le fait que la pile Bluetooth est externalisée dans une application système, l'application `bluego`.

[1] *SD Input/Output*, une extension du protocole de communication de cartes SD permettant d'intégrer d'autres interfaces (GPS, WiFi, Ethernet, Bluetooth etc.)

Cette application gère un modem Bluetooth externe au SoC applicatif et communique avec celle-ci par le biais d'une UART où transite le trafic à destination et en provenance de cette interface externe. Aucune donnée transitant par l'interface Bluetooth n'est traitée par le noyau. L'examen de l'application gérant le Bluetooth, `bluego`, montre que cette application fonctionne en utilisant la pile protocolaire `Blue SDK` de la société OpenSynergy.

2.2 Modèle d'attaquant

On trouve au sein des véhicules un connecteur de communication avec l'ensemble des ECUs présents, le connecteur *On-Board Diagnostics* (OBD). Il a pour fonction première de permettre à un personnel autorisé muni du logiciel adéquat de réaliser des opérations de maintenance sur le véhicule. On trouve en outre quelques connecteurs spécifiques, par exemple un connecteur USB relié à l'infodivertissement. Ainsi, un attaquant qui a pénétré à l'intérieur du véhicule se trouve en position d'interagir avec n'importe quel élément de celui-ci, *i.e.*, a *de-facto* la possibilité de réaliser l'attaque qu'il souhaite quitte à remplacer/altérer physiquement un ou plusieurs éléments du véhicule.

La compromission physique d'un véhicule a un impact dévastateur, et peu de moyens existent pour se protéger d'un attaquant présent à l'intérieur du véhicule. Toutefois, la sécurité périmétrique du véhicule rend impossible cette attaque (alarme, véhicule gardienné) ou fait qu'une attaque physique laisse des traces qui permettent de détecter ce type d'attaque (système de verrouillage opérationnel).

Modèle d'attaquant retenu

Le modèle d'attaquant retenu dans cette étude est celui d'un attaquant pouvant interagir avec le véhicule sans agir physiquement sur celui-ci.
L'attaquant sera en particulier en mesure d'intercepter ou manipuler des communications avec le véhicule. Il pourra aussi effectuer des communications de son propre chef avec le véhicule en utilisant des canaux légitimes.

Surface d'attaque. Compte tenu du modèle d'attaquant retenu, la surface d'attaque résiduelle du véhicule est réduite aux interfaces externes du véhicule. À titre d'exemple, sur le véhicule cible, on peut trouver entre

autre des capteurs de pression de pneus, des caméras avant et arrière, des capteurs de proximité, une interface Bluetooth, une interface WiFi et une interface GSM.

3 Bluetooth

La pile Bluetooth [8] constitue une architecture logicielle modulable permettant de transporter de la vidéo, de l'audio, des fichiers ou encore partager une connexion Internet. Elle est organisée en couches distinctes, permettant une abstraction entre les aspects matériels et logiciels, comme présenté dans la figure 1. Les couches inférieures, incluant la couche physique (RF) et le contrôleur (LMP et LL), nommées *Modem*, gère les transmissions radio-fréquences et les mécanismes bas-niveau, tandis que les couches supérieures, dites *Hôte*, prennent en charge des fonctions avancées telles que l'établissement des connexions, l'authentification et la gestion des profils applicatifs. L'interface entre les couches physiques et les couche supérieures est nommée *Host Controller Interface* (HCI). Cette structuration garantit une interopérabilité entre dispositifs de fournisseurs différents et assure une large adoption dans des domaines variés.

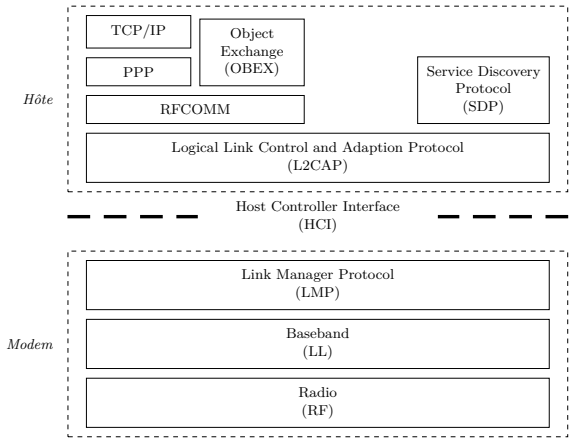

Fig. 1. Pile Bluetooth partielle.

Dans la partie *Hôte*, la couche Logical Link Control and Adaptation Protocol (L2CAP) joue un rôle clé dans la pile Bluetooth en servant de multiplexeur avec les protocoles des couches supérieures, comme Service Discovery Protocol (SDP) ou RFCOMM, et en assurant leur adaptation

pour une transmission efficace au-dessus de l'interface HCI. Elle permet notamment de configurer les canaux de communication en fonction des besoins spécifiques des protocoles applicatifs, en gérant des aspects comme la fragmentation et le multiplexage des données.

3.1 Service Discovery Protocol (SDP)

SDP est un protocole de type client/serveur utilisé dans les communications Bluetooth pour permettre à un appareil de *découvrir* les services offerts par un autre appareil Bluetooth. L'accès à ce type de services est réalisé dans une phase de pré-appairage, c'est-à-dire que *tous* les appareils à proximité, même ceux qui n'ont jamais été rencontrés jusqu'à présent, peuvent effectuer ces requêtes et qu'aucune validation utilisateur des périphériques effectuant les requêtes n'est requise. SDP permet de faciliter l'établissement d'une connexion en fournissant des informations détaillées sur les services disponibles, notamment leurs caractéristiques, attributs, et points de terminaison. Chaque service est décrit sous forme d'un enregistrement de service, structuré en une série d'attributs (paires clé-valeur), qui contiennent des informations telles que identifiant unique (UUID) du service, ses paramètres de communication et d'autres métadonnées. La figure 2 présente un échange SDP entre un téléphone et un casque audio.

Pour lister les services supportés par un périphérique Bluetooth, le protocole SDP défini deux types de commandes :

Recherche de services : Le client envoie une commande `ServiceSearchRequest`, qui inclut les types de services recherchés [8]. Le serveur retourne une réponse de type `ServiceSearchResponse`, comprenant une liste des attributs correspondant aux services supportés. Ces attributs sont des UUIDs permettant au client de continuer la consultation.

Récupération des attributs de service : Pour récupérer des attributs associés à un service spécifique, le client envoie une commande de type `ServiceAttributeRequest` en indiquant le handle reçu précédemment ainsi qu'une liste des attributs recherchés.

La réponse, de type `ServiceAttributeResponse`, contient les paires clé-valeur des attributs associées pour le service demandé. Ces informations peuvent être utilisées par les appareils pour sélectionner les services dont ils ont besoin et s'y connecter.

Pour accélérer les échanges, le client peut utiliser une commande de type `ServiceSearchAttributeRequest` qui combine la recherche de services et la récupération des attributs.

Dans la suite de cet article, nous nous focaliserons sur cette commande. Dans la figure 2, les échanges sont réalisées avec les commandes `ServiceSearchAttributeRequest` et les réponses `ServiceSearchAttributeResponse`.

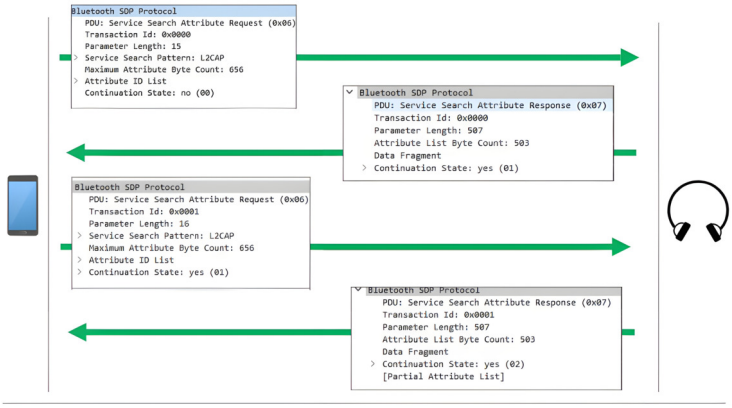

Fig. 2. Échanges entre un client (téléphone) et un serveur (casque) SDP ; inspiré de [7]. Cet échange, pré-appairage entre le casque et le téléphone, est nécessaire de savoir que c'est un périphérique audio, la présence de boutons, d'un micro, et potentiellement d'autres fonctionnalités.

La commande `ServiceSearchAttributeRequest` contient deux champs qui sont d'intérêt dans cet article :

`Maximum Attribute Byte Count` : en plus de la fenêtre de données recevables via la configuration du champ `MTU` dans la couche L2CAP, le champs `Maximum Attribute Byte Count` indique la taille maximale, en octets, des données d'attribut que le client est capable de recevoir dans une réponse. Ce champ permet de s'assurer que les réponses du serveur ne dépassent pas la capacité de traitement ou de mémoire du client.

`Continuation State` : est utilisé pour gérer les réponses partielles lorsque les données à transmettre dépassent la taille maximale spécifiée dans le champ `Maximum Attribute Byte Count` ou la `MTU` dans la couche L2CAP. Il contient un identifiant permettant au client de demander les données restantes lors de requêtes ultérieures, garantissant ainsi une transmission fragmentée et ordonnée.

Le `Continuation State` est composé de deux champs, `count` et `continuation_state_info`. Le champs `count` indique la taille

du champs `continuation_state_info` en octet. Le champs `continuation_state_info` est une suite d'octets à renvoyer lors de la requête suivante pour avoir la suite de ce paquet.

3.2 Implémentation de la pile Bluetooth dans notre cible

L'implémentation de la couche *Hôte* de pile Bluetooth embarquée dans notre cible est réalisée par la bibliothèque propriétaire `Blue SDK`.[2] Cette bibliothèque est en source fermée et peu d'informations publiques sont disponibles. Néanmoins, une CVE de 2018, la `CVE-2018-20378` [7], présente une possible exploitation d'un *buffer overflow*.

3.3 CVE-2018-20378

La `CVE-2018-20378`, également connue sous le nom de vulnérabilité « *Hell2CAP* » [7] concerne `Blue SDK` de la version à 3.2 à 6.0. Cette vulnérabilité permet à des attaquants à distance et non authentifiés d'exécuter du code arbitraire en exploitant un *buffer overflow* causé par une mauvaise gestion de la configuration de la `MTU` dans la couche L2CAP.

Dans cette section, nous décrivons les trois étapes menant à leur exploitation comme décrit dans le `CVE-2018-20378` : la mauvaise configuration du champs `MTU` dans la couche L2CAP, le *buffer overflow* et la prise de contrôle d'un pointeur de fonction (PFN) permettant l'exécution d'une charge malveillante.

Mauvaise configuration du champs `MTU` dans la couche L2CAP. L2CAP gère la signalisation et la configuration de canaux pour la communication Bluetooth. Lorsqu'une requête de configuration champs `MTU` arrive, `Blue SDK` affecte la valeur, en octet, demandée dans la structure représentant le canal de communication PUIS vérifie si elle est valide. Cette valeur permet de définir la taille des paquets de réponse encapsulés des couches supérieures.

Si la valeur est inférieure à la taille minimale spécifiée [8], qui est de 48 octets, le canal est marqué comme invalide, mais pas fermé. Sur ce point, la spécification Bluetooth [8] n'indique pas de comportement attendu. Si le client envoie par la suite des requêtes valides, le canal sera utilisable malgré une valeur `MTU` invalide. Cette vulnérabilité permet à un attaquant de contourner les contraintes de la spécification et de manipuler un tampon de réponse de taille non standard. La figure 3 présente un scénario pour configurer la `MTU` de la couche L2CAP à 20 octets.

[2] Pour plus d'information : `https://www.opensynergy.com/blue-sdk/`

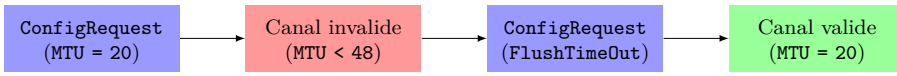

Fig. 3. Scénario de compromission de la valeur du champs `MTU` lors de la configuration de la couche L2CAP.

Écriture dans le *buffer overflow*. La couche SDP, utilise la configuration `MTU` que la couche `L2CAP` lui indique : elle utilise cette valeur pour déterminer le placement en mémoire des champs constituant sa réponse et pour décider de fragmenter ses paquets en conséquence.

Selon les informations partagées avec la `CVE-2018-20378`, la taille du paquet de réponse à une requête SDP est calculée en soustrayant 9 de la `MTU` du client (ligne 4, listing 1). Cette constante représente la taille, dans le paquet de réponse, de l'entête SDP. Si la `MTU` est inférieure à 9 octets, un *integer underflow* se produit, impactant la taille maximale du paquet de réponse.

Listing 1: Code source fourni par la `CVE-2018-20378` mettant en évidence l'*integer underflow* (en rouge).

```
1 // Dans le fichier core/stack/sdp/sdpserv.c
2 void SdpServHandleServiceSearchAttribReq (/* ... */ )
3   MTU = L2CAP_GetTxMtu(_sdpInfo->CID);
4   avalaibleSizeForFragment = (MTU - 9) & 0xFFFF;
5   // ...
6   // Construction de la réponse à la requête SDP
7   SdpStoreAttribData(_sdpInfo, _txPkt, _txPkt->buffetPtr,
8                      avalaibleSizeForFragment);
9 /* ... */ }}
```

Du *buffer overflow* à la modification d'un PFN. L'implémentation de `Blue SDK` peut supporter plusieurs clients SDP connectés simultanément. D'après les auteurs de la `CVE-2018-20378`, chaque client est associé à une structure SDP. Ces structures sont contiguës en mémoire et suivies par un PFN. Un *buffer overflow* depuis le dernier client pourrait donc atteindre ce pointeur, comme indiqué dans la partie haute de la figure 4.

En exploitant cela, l'attaquant peut réécrire le PFN avec une valeur choisie depuis le *buffer overflow* de réponse ; *c.f.* partie basse de la figure 4. Pour cela, les auteurs suggèrent l'usage de la structure `Continuation State` en l'alignant sur le mot à écrire.

Dans l'implémentation de `Blue SDK`, le champ `continuation_state_info` est encodé sur 1 octet. Il peut donc

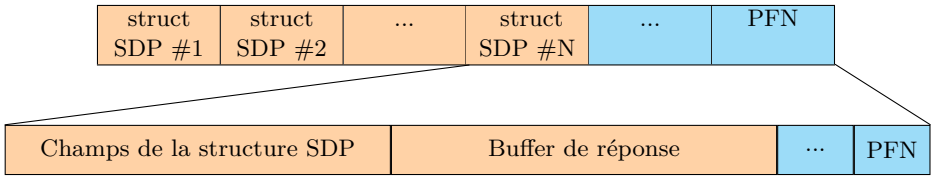

Fig. 4. Organisation mémoire selon la `CVE-2018-20378`.

prendre des valeurs comprises entre 1 et 255. En alignant ce champ sur l'octet à écrire, il devient possible de modifier sa valeur. Pour écrire un 0, il est nécessaire d'aligner le champ `count` sur l'octet du PFN à écrire.

Lorsque le PFN corrompu est appelé, le flot de contrôle est transféré à un emplacement maîtrisé par l'attaquant, exécutant sa charge malveillante. D'après les auteurs de la `CVE-2018-20378`, la cible étudiée est basée sur une architecture ARM 64 bits avec une DEP empêchant les données d'être exécutées. Ils ont donc fait du *Jump-Oriented Programming* (JOP). En revanche, aucun détail supplémentaire n'est donné sur l'exploitation réalisée.

Analyse de la `CVE-2018-20378`. Pour exploiter cette vulnérabilité, il est nécessaire de disposer de N clients SDP connectés simultanément. Le dernier client est celui qui déclenche le *buffer overflow* et modifie la valeur du PFN. Maintenir autant de clients connectés simultanément requiert une gestion fine de chaque client afin de conserver la connexion, ainsi que, du côté serveur, la capacité à supporter un nombre élevé de connexions en parallèle.

4 MTU et overflows

Dans la section précédente, nous avons présenté et analysé les détails publics de la `CVE-2018-20378`. À présent, nous étudions la sensibilité de notre cible à cette CVE ainsi que les conséquences potentielles de cette vulnérabilité, dans le cas où la cible serait effectivement vulnérable.

La figure 5 présente une vue d'ensemble du travail introduit dans cet article. Ce travail a été réalisé en boîte noire, avec toutes les fonctions de sécurité activées.

Pour confirmer la présence de la `CVE-2018-20378`, nous avons utilisé `Scapy` pour forger des paquets L2CAP comportant des configurations incorrectes. L'utilisation d'autres outils exploitant l'implémentation de la pile Bluetooth, tels que ceux basés sur BlueZ ou le noyau Linux, n'était

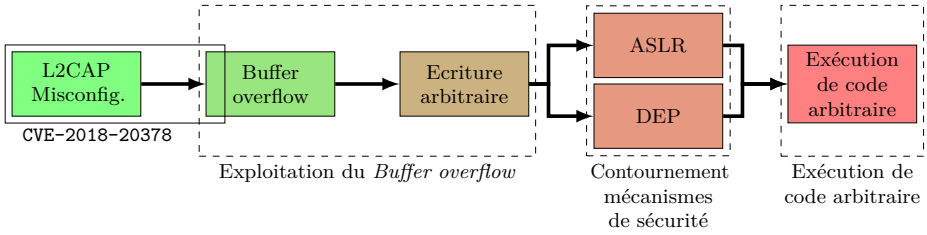

Fig. 5. Contributions présentées dans cet article : les contributions introduites dans cet article sont mises en évidence à l'aide de rectangles en pointillés, accompagnés d'un label décrivant chaque étape de ce travail.

pas envisageable, car ces outils imposent des paramètres conformes à la spécification Bluetooth [8].

Afin d'éviter toute interférence avec le système via l'interface Bluetooth, nous nous sommes interfacés directement au niveau HCI et avons implémenté la gestion de toutes les couches jusqu'à SDP. Cela a été rendu possible en étendant `Scapy` pour inclure des formats de paquets non encore supportés.[3] Grâce à cette approche, nous avons confirmé qu'il est possible de négocier une `MTU` dont la valeur est inférieure à 48, comme décrit dans la section 3.3. Cette observation confirme la présence de la vulnérabilité.

Pour décrire notre contribution, nous avons divisé l'attaque en trois étapes, représentées par les rectangles en pointillés dans la figure 5 :

1. **Exploitation du *buffer overflow*** (sous-section 4.2) : Nous étudierons d'abord le déclenchement du *buffer overflow* lors de la construction du paquet SDP de réponse, puis la création d'une primitive d'écriture arbitraire en mémoire.
2. **Contournement des mécanismes de protection** (sous-section 4.3) : La plateforme cible embarque des protections contre l'exécution de code arbitraire, telles que ASLR ou DEP, correspondant aux bonnes pratiques. Nous verrons que, si mal utilisées, elles peuvent être contournées.
3. **Exécution de code arbitraire** (sous-section 4.4) : Une fois les mécanismes de sécurité contournés, nous montrerons comment exécuter un code arbitraire.

Avant d'exploiter le *buffer overflow* résultant de l'*integer underflow* causé par le champ `MTU` dans la couche L2CAP, il faut comprendre l'implémentation de la pile Bluetooth.

[3] Un *pull request* sera créé sur le GitHub de `Scapy` pour la publication de cet article.

4.1 Préliminaires : analyse du fonctionnement de l'implémentation de la pile Bluetooth

Un paquet Bluetooth arrivant sur l'infodivertissement est d'abord réceptionné par le modem, puis transmis via une UART dédiée vers le processeur i.MX 6. Un *thread* spécifique de l'application `Bluego` reçoit le paquet et le place dans une file d'évènements en attente de traitement. D'autres *threads* se chargent ensuite de traiter ces évènements et de générer les réponses adéquates.
La pile Bluetooth et l'ensemble du traitement associé sont implémentés dans la bibliothèque `libbluego.so`, qui embarque `Blue SDK`.

Implémentation du serveur SDP. En analysant la bibliothèque `libbluego.so`, nous observons qu'un tableau statique de 7 entrées est alloué pour chaque protocole Bluetooth au-dessus de la couche HCI, dans un espace mémoire nommé `bt`. Ce tableau contient, pour chaque index, une structure représentant les informations associées à chaque protocole. Pour SDP, le tableau est nommé `sdpServer.infos` (voir listing 2, ligne 3) et contient des entrées de type `sdpServInfo`, associées à chaque client connecté au serveur SDP, notamment celles mentionnées dans la `CVE-2018-20378`.

Listing 2: Implémentation du serveur SDP en mémoire

```
1  bt { // ..
2    sdpServer = {
3      infos[7] = {
4        struct sdpServInfo[0] {
5  /* 0x0000 */ struct sdpServInfo * next;
6  /* 0x0004 */ struct sdpServInfo * prev;
7  /* 0x0008 */ uint8_t * ptr_pkt_data;
8    // ...
9  /* 0x0060 */ struct sdpAttribInfo *sdpAttrib;
10 /* 0x0064 */ struct sdpRecordInfo *sdpRecords;
11   // ...
12 /* 0x0088 */ uint8_t pkt_header [5];
13 /* 0x008D */ uint8_t pkt_data [507];
14   }; // taille = 648 (0x288) octets
15       // ... Description des instances de sdpServInfo 1 à 6
16     }
17 /* 0x11D6 */ uint32_t sémaphore;
18   } sdpClient {
19 /* 0x11DA */ void (*protocol_callback) (uint16_t, void *);
20 /* ... */ } };
```

La structure `sdpServInfo`, à partir de la ligne 4, est une liste doublement chaînée faisant référence à l'entrée précédente et suivante dans le tableau `infos`. Le pointeur `ptr_pkt_data`, ligne 7, fait référence au tableau `pkt_data`, situé en fin de structure, ligne 13, et contenant uniquement les données générées suite à une requête SDP. L'entête associée est stocké dans le tableau `pkt_header`.

Ligne 19, nous observons la présence d'un PFN nommé `protocol_callback`. Ce PFN est une *callback* appelée lorsque l'infodivertissement agit en tant que client SDP pour récupérer les fonctionnalités supportées par un périphérique auquel il souhaite se connecter. À la ligne 17, un sémaphore encodé sur 4 octets indique, par une valeur non nulle, qu'une réponse SDP est en cours de construction. Cela met en attente les autres requêtes au serveur SDP.

La présence de ce sémaphore rend l'exploitation décrite dans `CVE-2018-20378` impossible sur notre cible. Un débordement de tampon visant à modifier le PFN écraserait ce sémaphore avec des valeurs non maîtrisées, ce qui rend extrêmement complexe la modification de ce sémaphore avec une valeur nulle, rendant ainsi muet le service SDP.

Analyse du déclenchement de l'*integer underflow*. Dans la sous-section 3.3, nous avons présenté l'*integer underflow* tel qu'introduit dans la `CVE-2018-20378`, où aucune limite n'était spécifiée concernant la taille du dépassement. Le listing 3 montre une partie de la construction de la réponse à une requête SDP sur notre cible.

Listing 3: Pseudo-code présentant l'*integer underflow*, ligne 4 en rouge.

```
1 void SdpServHandleServiceSearchAttribReq ( /* ... */ ) {
2   // ...
3   taille_paquet_max = MIN (L2CAP_GetTxMtu(current_channel), 512);
4   taille_paquet_max = (taille_paquet_max - 9) & 0xFFFF;
5   taille_paquet_max =
6        MIN (rx_pkt->MaximunAttributeByteCount,
        ↪  taille_paquet_max);
7   // ...
8   taille_paquet_reponse = MIN(taille_paquet_max, taille_reponse);
9 /* ... */ };
```

Dans cette implémentation (listing 3), la taille maximale du paquet de réponse est déterminée par plusieurs facteurs, comme indiqué ci-dessous :

— la valeur minimale entre la `MTU` et 512 (ligne 3) ;
— la `MTU` diminuée de 9 octets (ligne 4) ;

— la valeur du champ `Maximum Attribute Byte Count`, spécifiée par le client dans la requête SDP (ligne 6) ;
— ou la taille totale de la réponse à la requête SDP (ligne 8).

4.2 Partie 1 : exploitation du *buffer overflow*

Grâce à `Scapy`, nous nous connectons à la cible et configurons la `MTU` du canal L2CAP à 8. Cette configuration déclenche un *integer underflow*, ce qui forge une taille de réponse maximale limitée par le minimum entre 65 535 octets et le champ `Maximum Attribute Byte Count` de la requête SDP. Avec une `MTU` de 8, et indépendamment de la valeur du `Maximum Attribute Byte Count`, la cible construit une réponse à la requête SDP, mais ne l'envoie pas. Un test effectué sur l'espace disponible pour les entêtes SDP empêche l'émission du paquet.

Déclenchement du *buffer overflow*. Dans l'implémentation cible, le tampon de réponse est alloué statiquement à 512 octets : 5 octets sont réservés pour l'entête, correspondant au tableau `pkt_header`, et 507 octets pour les données, correspondant au tableau `pkt_data`, dans listing 2. Avec une `MTU` de 8 et un `Maximum Attribute Byte Count` supérieur à 507, un *buffer overflow* se produit, contrôlé en taille par l'attaquant.

Pour notre cible, si nous envoyons la commande `ServiceSearchAttributeRequest` associée aux attributs du service L2CAP, la réponse est de 729 octets. Avec une valeur de `MTU` incorrecte, il est ainsi possible de faire un *buffer overflow* de 222 octets $(729 - 507)$.

Modification de l'adresse du pointeur de fonction. En reprenant le listing 2, nous choisissons de faire déborder le *buffer overflow* pour recouvrir le champs `ptr_pkt_data` de `sdpServInfo[1]` afin de compromettre l'emplacement des données de réponse d'un second client.

À l'instar de l'approche présentée dans la `CVE-2018-20378`, en faisant déborder le *buffer* de réponse sur le champ `ptr_pkt_data`, il est possible de modifier sa valeur. Pour cela, nous exploitons l'élément `Continuation State` de la réponse à une requête SDP, qui est écrit en dernier. Lorsque le champ `continuation_state_info` est non nul (`count` > 0), cela indique qu'il reste des données à transmettre. Sinon, `count` est nul.

L'implémentation cible ne supporte qu'un `continuation_state_info` encodé sur 1 octet, qui fonctionne comme un compteur monotone allant de 1 à 255, et dont la valeur est donc prédictible.

Pour écrire un 0, il faut aligner le champ `count` du dernier paquet de la réponse avec l'octet à écrire. Ce champ devient égal à 0 lorsque la transmission des données est terminée.

En jouant sur la taille des paquets précédents et la taille totale de la réponse, et en connaissant la valeur à écrire, il est possible d'ajuster l'état de `Continuation State` pour aligner `count` ou `continuation_state_info` sur l'octet cible.

Cependant, en raison de la structure du *buffer* de réponse, le champ `Continuation State` est toujours précédé d'au moins 16 octets. Ainsi, pour écrire 1 octet, il est nécessaire de modifier ces 16 octets.

Écriture arbitraire en mémoire. Une fois en capacité de modifier la valeur du pointeur de données `ptr_pkt_data`, nous pouvons réécrire n'importe quel emplacement en mémoire.

Pour réussir une telle écriture, nous utilisons 2 clients, nommés Client 1 et Client 2, qui réaliseront simultanément des requêtes Bluetooth ; le Client 1 déclenchera le *buffer overflow* et modifiera la valeur du pointeur `ptr_pkt_data` du Client 2. Le Client 2 devra forger des paquets choisis afin d'écrire les données voulues en mémoire.

Pour écrire ces données en mémoire, le Client 2 utilisera le champs `Continuation State`, comme vu précédemment. En revanche, contrairement au Client 1, le Client 2 n'a pas besoin de réaliser de débordement de tampons et écrit dans la taille de son *buffer* légitime.

Ces deux clients seront utilisés dans la suite de l'article sous ce nom.

4.3 Partie 2 : contournement des mécanismes de sécurité

La cible embarque tous les mécanismes de protection en profondeur attendus sur ce type de produit : ASLR et DEP. L'activation et la configuration du DEP est réalisée via la fonction `mprotect` embarquée dans la `libc`. Le moyen le plus simple de désactiver le DEP sur une partie de la mémoire est d'appeler nous aussi la fonction `mprotect` en réaffectant les permissions. Pour pouvoir faire cela, nous devons d'abord faire face à ASLR qui empêche d'utiliser directement les techniques de réutilisation de code (ROP, JOP) car il n'est pas possible, normalement, d'anticiper l'adresses des gadgets à utiliser. L'ASLR est actif sur la cible étudiée. Cela signifie que les adresses de base des segments de code, à laquelle les bibliothèques sont projetées en mémoire et l'adresse de base la pile de chaque processus sont non prédictibles. Pour rappel, la cible est une architecture 32 bits et son noyau Linux 3.0.35. Nous présentons ici un rappel du fonctionnement de l'ASLR sur cette version du noyau.

Fonctionnement de l'ASLR dans le noyau Linux 3.0.35. Lorsqu'une application est exécutée, l'appel système `execve` est effectué. Cet appel système, à son tour, appelle la fonction `load_elf_binary` du noyau. Cette fonction est celle qui va *in fine* réaliser la projection en mémoire du fichier à exécuter et donc réaliser le placement aléatoire des différentes sections en mémoire. Dans le cas qui nous intéresse, le fichier étudié est la bibliothèque `libc.so`. Elle est projetée en mémoire par le programme `ld_linux` dans la zone dévolue aux projections des bibliothèques

Pour déterminer comment une bibliothèque dynamique est chargée en mémoire, il est essentiel d'analyser le code du noyau.[4] Dans ce code, la macro `TASK_SIZE` est utilisée pour définir l'adresse du segment de code. Cette macro permet d'obtenir la taille maximale de la plage d'adresses utilisateur en se basant sur la valeur de la constante `CONFIG_DRAM_SIZE`. Dans l'implémentation cible, sa valeur est `0x80000000` (2 Go).

La zone dévolue à l'espace utilisateur (*userland*) va donc des adresses 0 à `TASK_SIZE-1`. Pour les projections des bibliothèques dynamiques la base de chargement est placée arbitrairement à `0x40000000` puis un aléa y est ajouté. Dès lors, les bibliothèques dynamiques sont chargées les unes à la suites des autres à partir de cette adresse de base.

La bibliothèque `libbluego.so` a un traitement particulier. Elle est chargée dynamiquement lors de l'exécution de l'application Java `bluego`, lorsque celle-ci fait des appels à l'infrastructure d'exécution de code natif d'Android (NDK). Ainsi, la bibliothèque `libbluego.so` est projetée en mémoire par un appel à la fonction `System.load` qui place les objects projetés en mémoire dans un espace privé alloué dans le tas. Cet espace est différent de celui utilisé par le système pour charger en mémoire des bibliothèques dynamiques et donc l'aléa sur son adresse de base est différent de celui appliqué sur la base de chargement de bibliothèques dynamiques classiques. Ainsi, dans notre cas, le segment de code de la bibliothèque `libbluego.so` se trouve projeté à l'adresse de base `0x5b000000`. La fonction `arch_randomize_brk`, listing 4, est appelée pour fournir un aléa qui sera rajouté à cette adresse.

[4] Disponible ici : `https://github.com/boundarydevices/linux.git`

Listing 4: Fonction `arch_randomize_brk`, fichier `arch/arm/kernel/process.c`, du noyau Linux 3.0.35.

```
514 unsigned long arch_randomize_brk(struct mm_struct *mm)
515 {
516   unsigned long range_end = mm->brk + 0x02000000;
517   return randomize_range(mm->brk, range_end, 0) ? : mm->brk;
518 }
```

Pour contourner l'ASLR, il faut retrouver les aléas associés à celui du segment de code, celui de l'adresse de base de projection des bibliothèques dynamiques et celui associé au tas. D'après la ligne 516 du listing 4, l'aléa utilisé sur cette architecture est d'au maximum, `0x2000000`. Néanmoins, l'examen de la fonction `randomize_range`, listing 5, montre aussi que cette valeur maximum est tronquée pour préserver l'alignement par rapport à une frontière de page mémoire.

Listing 5: Fonction `randomize_range`, dans le fichier `drivers/char/random.c` du noyau Linux 3.0.35.

```
1342 unsigned long
1343 randomize_range(unsigned long start, unsigned long end, unsigned
     ↪  long len)
1344 {
1345   unsigned long range = end - len - start;
1346
1347   if (end <= start + len)
1348     return 0;
1349   return PAGE_ALIGN(get_random_int() % range + start);
1350 }
```

La macro `PAGE_ALIGN`, ligne 1349 du listing 5, aligne l'adresse passée en paramètre sur une frontière de page, dans notre cas de 4 kB. C'est-à-dire qu'elle met à 0 les 3 *nibbles* de poids faible de l'adresse donnée.

Ainsi, à l'examen de ce code, l'aléa existant se réduit à un espace de taille `0x2000` (soit 8192 possibilités) et qu'il n'a un effet que sur les bits 12 à 25. En tant normal, 8192 possibilités constituerait une protection assez faible. Néanmoins, dans notre cas, notre canal de communication est extrêmement lent, de l'ordre de quelques octets par seconde. En conséquence, effectuer 8192 tests est trop long pour être réaliste.

Détermination de la valeur de l'ASLR. Trouver rapidement l'aléa utilisé pour projeter la `libbluego` en mémoire est une condition *sine qua*

none de la réalisation pratique d'une attaque. Afin d'y parvenir, il est à noter que le processeur de la cible est utilisé en mode en *Little Endian*.[5]

Pour inférer la base d'adresse de la bibliothèque `libbluego.so` en mémoire, nous allons commencer par déterminer les bits constituant le *nibble* de poids fort du second octet de l'aléa ; c'est-à-dire les bits 12 à 15 de l'adresse cherchée. Une fois ces bits obtenus on sera en mesure de réaliser une fuite d'information qui permettra d'obtenir les autres bits d'adresse.

Détermination du nibble *de poids fort du second octet.* Deux conditions sont réunies pour déterminer la valeur *nibble* de poids fort du deuxième octet de l'aléa utilisé :

- Les 12 bits de poids faible des pointeurs ne sont pas impactés par l'ASLR.
- En mémoire, avant et après le tableau `SdpServer.infos`, ligne 3 du listing 2, se trouvent plusieurs PFNs qui sont appelées à différentes étapes du protocole Bluetooth. On en choisit un dont on pourra déclencher un déréférencement. Ce PFN nous servira alors d'oracle pour savoir si on a trouvé le bon aléa. Pour la suite de cette section, ce PFN sera nommé `PTR_ORACLE`.

Pour identifier la valeur du *nibble* de poids fort du deuxième octet de l'aléa, nous allons tester toutes les valeurs possibles (16 valeurs au total) de ce *nibble*. Pour tester une valeur possible il est nécessaire d'utiliser deux clients SDP, à l'instar de la sous-section 4.2. Le premier va, via le *buffer overflow*, modifier la valeur du pointeur `ptr_pkt_data` du Client 2 afin de le positionner à l'adresse où se trouve stockée `PTR_ORACLE` : on remarque que lors de l'initialisation de la bibliothèque `libluego.so`, le pointeur `ptr_pkt_data` de chaque `SdpServInfo` est assigné à l'adresse de son propre champ `pkt_data`. Ensuite, en examinant le binaire `libluego.so`, avant le chargement en mémoire et l'application de l'aléa, on constate que les 16 bits de poids fort de l'adresse du champs `pkt_data` du Client 2 sont les mêmes que ceux de l'adresse du pointeur `PTR_ORACLE`. En revanche, en considérant l'adjonction d'aléa sur ces deux adresses, les deux octets de poids fort de ces deux adresses peuvent différer. En fait, ils sont soit identiques, soit différents de 1 ; en fonction de la valeur du *nibble* de poids fort du deuxième octet de l'aléa. Statiquement, dans la `libbluego.so`, le *nibble* de poids fort de l'adresse de `ptr_pkt_data` du Client 2 vaut `0xE`, tandis que celui de `PTR_ORACLE` vaut `0xF`. Le décalage de 1 sur les octets

[5] En *Little Endian*, les données sont stockées en mémoire de l'octet de poids faible à l'octet de poids fort.

de poids fort se produit si l'addition de l'aléa entraîne une retenue sur ce *nibble* pour l'adresse de `PTR_ORACLE`, mais pas pour celle de `ptr_pkt_data`. Cela implique que ce scénario ne se produit que si la valeur de l'aléa sur ce *nibble* est 1.

En faisant abstraction ici de l'aléa, pour faire pointer le pointeur `ptr_pkt_data` sur `PTR_ORACLE`, il suffit uniquement de modifier ses deux octets de poids faible. A cause de l'*endianess*, le débordement de tampon permet de réaliser cette modification.

Dans le cas où le décalage de 1 se produit, `ptr_pkt_data` du Client 2 ne pointera pas vers le `PTR_ORACLE`.

Comme évoqué plus haut, les 12 bits de poids faible des adresses sont laissés inchangés par l'aléa de l'ASLR. L'analyse statique de la bibliothèque `libbluego.so` suffit donc à obtenir leurs valeurs pour le pointeur `PTR_ORACLE`, même après randomisation. On notera `0xXXX` la valeur de ces 3 *nibbles* de poids faible et n celle du *nibble* de poid fort du deuxième octet ; on a donc la valeur de 2 derniers octets est `0xnXXX`.

Pour déterminer la valeur de n, il est nécessaire de faire une recherche exhaustive. Tout d'abord, on positionne, via le *buffer overflow*, les deux octets de poids faible du `ptr_pkt_data` du Client 2 à la valeur `0xnXXX`.

Ensuite, le Client 2 effectue une requête SDP légitime qui écrit la réponse dans les données pointées par `ptr_pkt_data`, corrompant ainsi ces dernières (les valeurs écrites sont connues).

Enfin, si la valeur de n testée est correcte, alors le `ptr_pkt_data` du Client 2 pointe sur le `PTR_ORACLE`, corrompant sa valeur. Cette nouvelle valeur pointe vers une zone inaccessible de la mémoire. On envoie alors une requête déclenchant le déréférencement. L'utilisation de `PTR_ORACLE` provoque une erreur de segmentation et aucune réponse n'est reçue. En revanche, si la valeur testée est incorrecte, une réponse sera reçue.

Au final, tester toutes les valeurs de n correspond à au maximum 16 tests et cela prend environ une minute pour obtenir l'aléa.

Fuite d'information. La principale difficulté pour obtenir une primitive de lecture réside dans le fait que les données reçues par le client sont celles construite en réponse à une requête SDP. L'objectif est donc de faire en sorte que les données en réponse soient différentes de celles initialement écrites. En connaissant le *nibble* de poids fort du second octet de l'aléa, on peut exploiter le *buffer overflow* depuis le Client 1, une nouvelle fois, pour modifier les 2 octets de poids faible de la valeur pointée par le `ptr_pkt_data` du Client 2. Cette fois-ci dans l'objectif de le faire pointer

au milieu du `pkt_data` du Client 1. Le tableau `pkt_data` du Client 1 étant situé immédiatement avant la structure `sdpServInfo` du Client 2, le *buffer overflow* permet de réécrire les premiers champs de celle-ci. L'objectif est que le dernier octet de la réponse, le `Continuation State`, écrase l'octet de poids faible du `ptr_pkt_data` du Client 2 via une requête SDP légitime. Cette modification a lieu juste avant l'envoi de la réponse et fournit une réponse qui englobe une partie des champs de la structure `sdpServInfo` du Client 1. Comme visible dans le listing 2, deux pointeurs suivent `pkt_ptr_data` : `sdpAttribInfo` (ligne 9) et `sdpRecordInfo` (ligne 10). Ainsi, affecter grâce à la manipulation du `ptr_pkt_data` du Client 2, nous obtenons leurs valeurs.

Cette fuite d'information révèle l'intégralité de l'aléa via les adresses des pointeurs. Cependant, il est important de noter que cela ne constitue pas une primitive de lecture arbitraire, car seules les données adjacentes au tableau `pkt_data` du Client 1 peuvent être dévoilées dans ce cas.

4.4 Partie 3 : exécution de code arbitraire

Dans la sous-section 4.2, nous avons expliqué comment écrire où nous voulions dans la mémoire du processus lié à la `libbluego`. Dans la section précédente, sous-section 4.3, nous montrons qu'il est possible de déterminer la base d'adresse en mémoire de la bibliothèque `libbluego.so`. A présent, nous pouvons forger une adresse valide d'un PFN et prendre le contrôle du flot d'exécution.

Dans cette section, nous présentons, dans un premier temps, notre recherche d'un PFN modifiable qui serait facilement accessible par **un attaquant à distance sans intervention des occupants du véhicule**. Dans un second temps, il va nous falloir rediriger le flot de contrôle vers une zone mémoire exécutable que nous maîtrisons. Cela nécessite de rendre la mémoire exécutable.

Les aventuriers du PFN perdu. Lorsqu'une connexion L2CAP est initiée, elle est associée à un protocole définit par le champs *Protocol Service Multiplexing* (PSM) de la requête du client. Dans l'implémentation cible, la gestion des protocoles est associée à la structure `Protocol`, décrite dans le listing 6. Dans ce listing, les PFN sont en rouge.

Listing 6: Implémentation de la gestion des protocoles

```
1 struct Protocol {
2   void (*proto_callback)(uint16_t, void *);
3   uint16_t psm; // Type de protocole (1: SDP, 2: RFCOMM, ...)
4   uint16_t localMTU; // MTU vers le périphérique connecté
5   uint16_t remoteMTU; // MTU depuis le périphérique connecté
6   // ...
7   byte (*agent_allocator)(void);
8 /* ... */ };
```

Pour chaque connexion, deux *callbacks* sont utilisées. Une première fonction, nommée `proto_callback`, ligne 2 du listing 6, est associée à la gestion du protocole après toutes les étapes associées aux couches du modem jusqu'à L2CAP inclus.

Une seconde fonction, appelée `agent_allocator`, ligne 7 du listing 6, est chargée de gérer le cycle de vie des canaux via un agent dédié. Pour chaque canal, un agent est alloué lors de la connexion. Cet agent joue un rôle central en établissant un lien entre toutes les connexions associées au protocole, les requêtes des clients, et les ressources correspondantes. Ainsi, il garantit une gestion cohérente et unifiée des interactions.

La figure 6 illustre l'ordre chronologique d'appel des *callbacks* `proto_callback` et `agent_allocator` pendant l'établissement d'une connexion sécurisée.

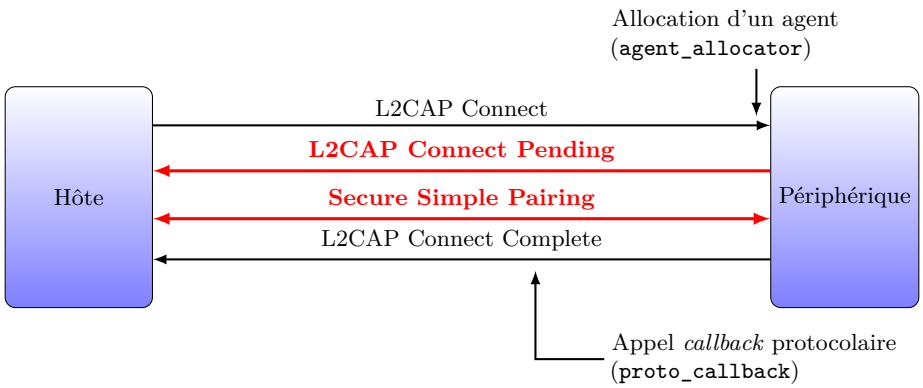

Fig. 6. Illustration de la connexion entre un hôte et un périphérique. Les flèches noires représentent une connexion classique établie via un canal Bluetooth non sécurisé. En revanche, dans le cas d'une connexion sécurisée, les flèches sont en rouge et en gras pour signaler les étapes supplémentaires impliquées dans le processus d'authentification.

La fonction `agent_allocator` est appelée dès la réception d'une demande de connexion, comme décrit dans la figure 6. Il est donc intéressant pour un attaquant de modifier le PFN associé afin de le corrompre, que la connexion soit sur un canal sécurisé ou non. Pour la suite, nous nous focaliserons sur les conséquences de la modification de ce PFN afin d'exécuter une *payload* **sans authentification ni interaction avec les occupants du véhicule cible**.

Exécution de code arbitraire. Pour contourner le mécanisme de DEP, nous devons appeler la fonction `mprotect`. Cette fonction est disponible dans la `libc` partagée par l'environnement d'exécution cible. Cette fonction n'est cependant pas directement résolue dans la bibliothèque `libbluego.so`, car `mprotect` n'est pas un symbole nécessaire à son exécution. Toutefois, la bibliothèque `libbluego.so` utilise des fonctions de la `libc`, ce qui signifie que la `libc` est entièrement accessible depuis l'espace mémoire de la `libbluego`. La `libc` est randomisée différemment de celle de `libbluego`.

Dans un fichier ELF, les symboles nécessaires lors de l'exécution dynamique sont résolus grâce à la Global Offset Table (GOT) et à la Procedure Linkage Table (PLT). Lors de l'exécution, la GOT contient les adresses des fonctions et des variables externes, qui sont mises à jour au besoin par le chargeur dynamique, permettant ainsi une résolution des symboles à la volée pendant l'exécution du programme.

Pour obtenir l'adresse de la base de la `libc`, il est donc nécessaire de récupérer l'adresse d'une fonction déjà résolue de la `libc` depuis la table GOT mappée en RAM. Grâce à des gadgets présents dans `libbluego.so`, nous pouvons construire un mécanisme permettant de lire une adresse depuis la table GOT. Cette suite de gadgets nous permet d'avoir une primitive nous permettant de lire une donnée arbitraire en mémoire. Cette information nous permet ensuite de déduire la base de la `libc` en mémoire et de calculer l'adresse de `mprotect`. Enfin, en utilisant une autre suite de gadgets disponible dans la bibliothèque `libbluego.so`, nous sommes en mesure d'appeler `mprotect` pour rendre exécutable une zone mémoire précédemment protégée, permettant ainsi de contourner le DEP et d'exécuter notre *shellcode* en RAM.

L'ensemble de cette attaque, incluant les écritures permettant d'exécuter les deux suites de gadgets et le *shellcode*, **est réalisé en moins de 300 secondes**.

5 Impact de sécurité

Une fois la capacité de prise de contrôle de l'infodivertissement via *shellcode*, nous nous sommes rapprochés du constructeur de la cible dans le cadre du processus de *responsible disclosure*. En effet, la bibliothèque `Blue SDK` contenant la vulnérabilité que nous exploitons est largement présente dans des véhicules de constructeurs et de modèles différents. Cette procédure a ainsi permis au constructeur d'engager les actions nécessaires en interne et auprès des partenaires/prestataires concernés afin d'analyser et de corriger la vulnérabilité exposée.

Le travail présenté dans cet article s'est déroulé entre septembre 2023 et avril 2024. Le constructeur du véhicule cible a été informé de la vulnérabilité en septembre 2024 et a été particulièrement réceptif et réactif. La présente publication a été autorisée en accord avec ce constructeur, à la condition de ne pas dévoiler d'informations permettant d'identifier le modèle ou la marque du véhicule. Par ailleurs, le constructeur précise que si l'attaque présentée permet l'exécution de commandes CAN depuis le *shellcode* avec des droits privilégiés, celles-ci sont filtrées, rendant impossible l'exécution de commandes critiques.

6 Travaux connexes et état de l'art

Dans le travail présenté dans cet article, nous nous sommes concentrés sur la sécurité du système d'infodivertissement automobile via ses modules de communication. Cette étude a mis en évidence que le Bluetooth pouvait être exploité comme vecteur d'attaque afin de prendre le contrôle de l'environnement d'exécution de l'infodivertissement. L'objectif de cette section est de proposer un aperçu à un plus haut niveau des travaux connexes et de l'état de l'art relatifs à la sécurité des véhicules connectés et autonomes.

Ces enjeux de sécurité au sein des véhicules sont actuellement pris en compte par différentes instances règlementaire, ainsi, depuis 2021, plusieurs réglementations des Nations Unies[6] traitent spécifiquement des enjeux liés à la cybersécurité automobile et à la gestion des mises à jour logicielles. Ces réglementations renforcent la prise de conscience des constructeurs quant à l'importance du maintien en condition de sécurité

[6] `R155` « *UN Regulation No. 155 : Cyber security and cyber security management system* » [1] et `R156` « *UN Regulation No. 156 : Software update and software update management system* » [2]

des véhicules connectés tout au long de leurs cycles de vie. Cette prise en compte a été provoquée par plusieurs attaques significatives rendues publiques ayant affectées la sécurité des véhicules connectés au cours des années précédentes.

Les sous-sections qui suivent présentent quelques unes de ces attaques en fonction des éléments du véhicule mis en défaut. Plus spécifiquement, la sous-section 6.1 présente les vulnérabilités identifiées au niveau des architectures déployées. Les attaques qui y sont exposées sont rendues possibles principalement du fait que l'architecture utilisée est *à plat* (ce qui n'est plus l'approche préconisée comme cela sera précisé dans la sous-section 6.6). La sous-section 6.2 décrit ensuite des vulnérabilités affectant l'infodivertissement. Cet élément, devenu l'interface principale d'interaction avec les occupants du véhicule, est également assujetti à des failles de sécurité pouvant compromettre la sécurité globale du véhicule – les travaux présentés dans cet article en sont une illustration flagrante. La sous-section 6.3 aborde ensuite les conséquences possibles d'une vulnérabilité affectant les mécanismes d'ADAS (*Advanced driver-assistance systems* ou aide à la conduite), avec des impacts à la fois sur la sécurité des conducteurs et sur celle des véhicules autonomes. La sous-section 6.4 expose ensuite les risques spécifiques de sécurité associés à l'utilisation de motorisations *électriques* dans les véhicules. Enfin, la sous-section 6.5 présente une synthèse et une analyse de l'état de l'art proposé, tandis que la sous-section 6.6 décrit plusieurs initiatives récentes permettant d'améliorer sensiblement la sécurité du bus CAN à travers la mise en place de mécanismes de ségrégation et le renforcement de la sécurité de l'environnement d'exécution.

6.1 Sécurité de l'architecture

Historiquement, les premiers travaux portant sur la sécurité des véhicules connectés se sont intéressés aux vulnérabilités liées à l'architecture interne des véhicules, en particulier à l'organisation et aux échanges entre les différents composants embarqués. Cela fait suite aux démonstrations réalisées par Miller et Valasek sur des véhicules de marque Jeep [25, 26]. En effet, ces véhicules exposaient sur leur interface GSM un service permettant d'envoyer directement des messages DBus vers le module d'infodivertissement. En raison de la topologie *à plat* du bus CAN utilisé, il a été possible pour ces deux chercheurs d'injecter des trames arbitraires sur le bus, provoquant ainsi les dysfonctionnements mis en évidence lors de leurs démonstrations.

Dans le même intervalle de temps, une publication au SSTIC 2016 par Pollet et Massaviol [30] décrivait une analyse réalisée sur plusieurs dispositifs d'infodivertissement. Cette étude pointait différents défauts identifiés durant l'analyse et mettait en lumière les évolutions nécessaires de ces dispositifs au regard des recommandations en matière de sécurité.

Plus récemment, une autre vulnérabilité majeure a été révélée chez Toyota [35]. Un défaut de conception dans l'organisation physique des bus CAN exposait directement celui auquel étaient connectés les calculateurs en charge de l'antidémarrage et de l'ouverture du véhicule. La connexion d'un boîtier externe imposant des niveaux électriques spécifiques sur les lignes du bus CAN permettait ainsi de simuler l'envoi de messages sur ce bus critique, contournant totalement les mécanismes de sécurité du véhicule.

En élargissant le spectre des menaces, il est pertinent de mentionner les travaux de Gardiner et al. [15], mettant en évidence une faiblesse physique dans la conception du bus interne de communication de certains poids lourds américains (norme `SAE J2497` [33]). Cette vulnérabilité permet à un attaquant situé à l'extérieur du véhicule d'injecter des commandes sur le bus desservant les remorques tractées. Cette faille provient de l'utilisation de la technologie courants porteurs en ligne (CPL), qui mutualise plusieurs canaux de communication sans recourir à des paires différentielles, ce qui rend ce système vulnérable aux injections électromagnétiques. Dans une approche similaire, les travaux de Colin O'Flynn [28] démontrent la susceptibilité de certains ECU aux injections électromagnétiques *in situ*, via des techniques réalisables avec des équipements « standards » accessibles à un garage automobile.

Enfin, comme la sécurité physique constitue une propriété essentielle pour les véhicules, le système de verrouillage représente une pierre angulaire de cette sécurité. De nos jours, rares sont les véhicules pour lesquels ce mécanisme repose uniquement sur une clé physique classique. Ainsi, divers protocoles d'authentification sans fil ont été développés, impliquant, comme trop souvent, des mécanismes cryptographiques conçus de façon *ad hoc*. Parmi les analyses notables, les publications sur le protocole Hitag2 [5, 14] démontrent les faiblesses inhérentes à ce protocole cryptographique. Ces attaques sont d'autant plus critiques qu'une fois que l'attaquant accède physiquement à l'intérieur du véhicule, il dispose également d'un accès au connecteur OBD, lui conférant alors des capacités d'attaque beaucoup plus importantes.

6.2 Sécurité de l'infodivertissement

Comme évoqué dans la sous-section précédente, l'architecture interne du véhicule présente de nombreuses vulnérabilités potentielles. Parmi ces composants, le système d'infodivertissement constitue un élément central, assurant l'interface principale d'interaction entre les occupants du véhicule et les différentes fonctions embarquées. Du fait de ce rôle pivot et des nombreux canaux de communication qu'il intègre, il constitue une cible privilégiée pour les attaquants.

Au-delà des travaux présentés dans cet article et parmi les publications récentes centrées sur la sécurité des systèmes d'infodivertissement, une attaque exploitant la connectivité Bluetooth des dispositifs embarqués de marque Alpine a été mise en évidence. Cette attaque consistait en une prise de contrôle non authentifiée du dispositif, basé sur une architecture ARM 32 exécutant un système Android sans mécanismes de sécurité tels que le *stack canary* ou l'application du *Position Independent Executable* (PIE). Une implémentation propriétaire de la pile Bluetooth était employée en lieu et place de la pile standard Linux ; comme dans le cas étudié dans le présent article. La vulnérabilité initiale exploitée était un *use-after-free*, situé dans la couche HCI, précisément dans la gestion des connexions asynchrones non authentifiées au niveau de la couche L2CAP. L'exploitation réussie de cette vulnérabilité a permis d'obtenir une exécution de code arbitraire non authentifiée et sans interaction de l'utilisateur.

Ce panorama des menaces doit également inclure l'attaque découverte par Synacktiv sur les véhicules Tesla, publiée simultanément à la précédente [10]. En détournant la connexion issue d'un capteur de pression des pneumatiques, les chercheurs sont parvenus à réaliser une exécution de code arbitraire sur le module principal d'infodivertissement du véhicule. Cette attaque souligne l'étendue de la surface d'attaque du véhicule, composée de nombreux canaux fortement mutualisés. Contrairement à l'attaque précédente, l'architecture interne des véhicules Tesla a permis aux attaquants de s'ouvrir un chemin d'attaque plus profond vers plusieurs autres ECUs.

D'autres publications, bien que moins médiatisées, mettent également en lumière un nombre conséquent de vulnérabilités existantes au sein du parc automobile actuel. Citons par exemple l'attaque Koffee [9], qui consistait à injecter une application malveillante (*apk*) sur un dispositif d'infodivertissement préalablement intègre. Cette application malveillante communiquait ensuite avec le démon en charge des échanges sur le bus

CAN, appelé `micomd`, permettant ainsi d'injecter arbitrairement des trames sur le bus CAN dédié au divertissement.

6.3 Sécurité des ADAS et des composants tiers

Les attaques évoquées dans les paragraphes précédents concernent principalement des fonctions classiques des systèmes d'information. Toutefois, le domaine automobile présente une spécificité notable, que l'on retrouve également dans l'avionique et les systèmes industriels : les problèmes liés à la sûreté de fonctionnement deviennent également des problèmes de sécurité, car toute atteinte à une fonction de sûreté peut directement compromettre la sécurité physique des personnes.

Dans cette optique, Cao et al. [6] mettent en évidence la possibilité pour un attaquant distant d'altérer la perception des radars embarqués dans les véhicules autonomes, en supprimant virtuellement la détection d'éléments physiques présents dans leur environnement immédiat. Cette attaque, bien que n'impliquant pas directement les systèmes informatiques internes du véhicule, peut avoir un impact majeur sur la confiance placée dans la conduite autonome.

En complément, les travaux de Petit et al. [29] démontrent la possibilité inverse : créer de faux obstacles en perturbant les capteurs du véhicule. Les auteurs présentent ainsi des dispositifs capables de tromper des systèmes LiDAR (*Light Detection And Ranging*) et caméras en générant artificiellement des obstacles inexistants, provoquant une immobilisation complète du véhicule. Dans cette même étude, les chercheurs montrent également comment l'émission contrôlée d'un faisceau laser peut perturber les systèmes LiDAR et les caméras embarquées, rendant des personnes ou des objets réels totalement indétectables pour les systèmes de perception du véhicule, ce qui pourrait provoquer des accidents graves.

Par ailleurs, Kreil et al. [24] révèlent qu'un simple plot physique positionné devant un véhicule autonome peut suffire à rendre ce dernier totalement inopérant. En effet, bien que les capteurs détectent cet obstacle mineur, le système de conduite autonome n'est parfois pas en mesure de déterminer une stratégie permettant de contourner ou d'éviter l'objet, bloquant ainsi le véhicule sur la voie.

Enfin, une vulnérabilité [32] plus récente affectant les véhicules Kia a démontré la possibilité d'ouvrir un véhicule en exploitant une infrastructure externe (ou débarquée) du constructeur, simplement à partir d'informations inscrites sur la plaque d'immatriculation. Ce type d'attaque souligne l'interdépendance croissante entre le véhicule et les infrastructures externes des constructeurs, où une faille dans l'infrastructure distante peut

avoir un impact direct sur la sécurité physique des véhicules, sans nécessiter de compromettre directement les composants internes du véhicule.

6.4 Sécurité des véhicules connectées électriques

Afin de réduire les émissions de CO_2, une proportion croissante des véhicules est équipée d'une motorisation électrique. Contrairement aux moteurs thermiques, ce type de motorisation nécessite un mode de recharge spécifique, installé à domicile ou accessible via des infrastructures partagées, telles que les bornes situées dans les parkings de centres commerciaux ou sur les aires d'autoroutes. Des solutions innovantes permettent également la recharge dynamique lorsque les véhicules circulent sur des voies spécialement aménagées [27].

Dans ce contexte, l'architecture *Vehicle-to-Grid* (V2G) permet une interaction bidirectionnelle entre les véhicules électriques et le réseau électrique. Ainsi, en plus de la recharge classique des batteries, le V2G autorise les véhicules à restituer de l'électricité au réseau lors des pics de consommation. Cette technologie améliore significativement la flexibilité du réseau, facilitant notamment l'équilibrage de la demande énergétique et l'intégration efficace de sources renouvelables intermittentes comme l'éolien ou le solaire.

Cependant, cette interconnexion entre les véhicules électriques et le réseau introduit des enjeux de sécurité critiques, particulièrement au niveau des modules de charge. Ceux-ci représentent des vecteurs d'attaque potentiels pouvant compromettre simultanément la sécurité des véhicules et celle des infrastructures énergétiques.

Plusieurs travaux soulignent ces enjeux, notamment liés à la complexité des protocoles utilisés pour mettre en œuvre le V2G. Pour permettre l'échange de données entre les véhicules et les unités de recharge, les communications reposent généralement sur des transmissions en CPL. Comme indiqué dans [22], les systèmes V2G reposent sur une succession complexe de protocoles, parfois mal spécifiés ou mal implémentés.

En 2019, Dudek et al. [11] ont présenté à SSTIC l'outil *V2G Injector*, démontrant qu'en l'absence de mécanismes assurant l'authenticité des messages, un attaquant connecté à une borne compromise pouvait interagir avec des véhicules ou d'autres bornes légitimes via la ligne électrique. Il devient alors possible d'injecter des messages malveillants, par exemple pour empêcher un véhicule de se recharger.

Un état de l'art complet sur la sécurité des architectures V2G est proposé dans [22]. Les auteurs insistent sur la nécessité d'adopter une

approche globale de sécurité, intégrant l'ensemble des couches, depuis les protocoles de communication jusqu'aux infrastructures de recharge. En complément, Ali Sayed et al. [34] démontrent que certaines attaques ciblant les véhicules électriques peuvent engendrer des perturbations importantes sur le réseau électrique lui-même, menaçant ainsi la stabilité de l'approvisionnement énergétique. Enfin, des cas réels documentés dans [16] illustrent l'infiltration effective de bornes de recharge par des acteurs malveillants, soulignant l'urgence d'améliorer significativement la sécurité des modules de charge, quel que soit leur contexte de déploiement (domicile, espace public ou itinérant). Dans la même veine, il a été récemment démontré les capacités d'intrusions sur les interfaces offertes par les stations de recharge en exploitant les câbles de recharge [36].

6.5 Analyse de l'état de l'art des attaques

Les travaux présentés dans cette section révèlent l'ampleur des vulnérabilités affectant les véhicules connectés, qu'il s'agisse de l'architecture interne, des modules d'infodivertissement, des systèmes ADAS ou encore des composants tiers. Dans le cas particulier des véhicules électriques, la sécurisation des modules de recharge apparaît comme un maillon essentiel dans la chaîne de protection. La compromission de ces interfaces n'impacte pas seulement la sécurité individuelle d'un véhicule, mais peut également entraîner des perturbations à plus grande échelle, notamment sur le réseau électrique global. Il devient ainsi indispensable de mettre en œuvre des stratégies de défense robustes, incluant des mécanismes d'authentification renforcée, des protocoles de communication sécurisés et une surveillance continue des infrastructures de recharge, afin de garantir une sécurité *end-to-end* dans un écosystème automobile de plus en plus interconnecté.

Les problématiques liées aux ECUs et à leurs interconnexions ne sont pas propres au domaine automobile. Elles se retrouvent également dans d'autres secteurs industriels, comme l'agriculture ou les transports ferroviaires. À ce titre, les agriculteurs américains [21] ont été parmi les premiers à expérimenter les conséquences de la dépendance vis-à-vis des constructeurs : certains tracteurs embarquent des mécanismes bloquants dans les ECUs qui imposent une maintenance exclusive en atelier sous peine d'immobilisation. Des cas similaires ont été observés dans le domaine ferroviaire [19, 31]. Dans [31], un verrou logiciel intégré sciemment dans l'ECU impose un passage régulier en atelier constructeur pour éviter l'arrêt du train. Dans [19], une faiblesse dans l'authentification d'un message prioritaire permettait à un attaquant distant de provoquer un arrêt d'urgence de certains trains, entraînant la paralysie partielle du

trafic ferroviaire polonais. Ces exemples soulignent la transversalité des enjeux de sécurité liés aux ECUs et la nécessité de concevoir des systèmes résilients, y compris dans des environnements critiques, face à des menaces toujours plus complexes et interconnectées.

6.6 Initiatives pour sécuriser les véhicules

Les travaux présentés précédemment mettent en évidence la nécessité de renforcer la sécurité des systèmes embarqués dans les véhicules connectés, notamment en ce qui concerne la protection du bus CAN et des environnements d'exécution. Afin de répondre à ces enjeux critiques, plusieurs initiatives ont émergé, proposant des approches techniques et organisationnelles visant à limiter les risques de compromission et de propagation des attaques.

Ainsi, pour limiter le risque de propagation lorsqu'un ECU est compromis, il est nécessaire de ségréger le réseau CAN sur lequel chaque ECU est interconnecté [20]. La figure 7 présente une vue schématique de cette ségrégation, dans laquelle une *gateway* contrôle l'ensemble des échanges entre les sous-réseaux « *Habitacle* », « *ADAS* » et « *Motorisation* ». Si cette *gateway* permet effectivement d'empêcher la transmission de commandes non autorisées, elle constitue désormais une cible privilégiée pour les attaquants [13], soulignant ainsi l'importance critique de sécuriser cet élément central.

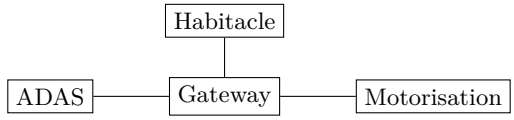

Fig. 7. Architecture où le réseau CAN est ségrégué. Adapté de [20].

En parallèle à une sécurisation de l'architecture de communication, l'initiative AUTOSAR[7] (*AUTomotive Open System ARchitecture*) vise à standardiser l'architecture logicielle des ECUs, en facilitant l'interopérabilité entre les constructeurs et les fournisseurs. La plateforme AUTOSAR Classic [4], conçue pour les systèmes embarqués temps réel, repose sur une architecture logicielle en couches, composée de trois niveaux principaux, comme présenté dans la figure 8 :

[7] `https://www.autosar.org`

— **La couche applicative** : contient les composants logiciels applicatifs, indépendants du matériel, qui implémentent les fonctionnalités spécifiques du véhicule.
— **L'environnement d'execution** : assure la communication entre les composants applicatifs et les services systèmes.
— **Les services systèmes** : fournit des modules logiciels standardisés pour l'environnement d'execution (comme la gestion de la mémoire, la communication et l'abstraction matérielle) permettant ainsi une réuilitisation et une portabilité accrues du code.

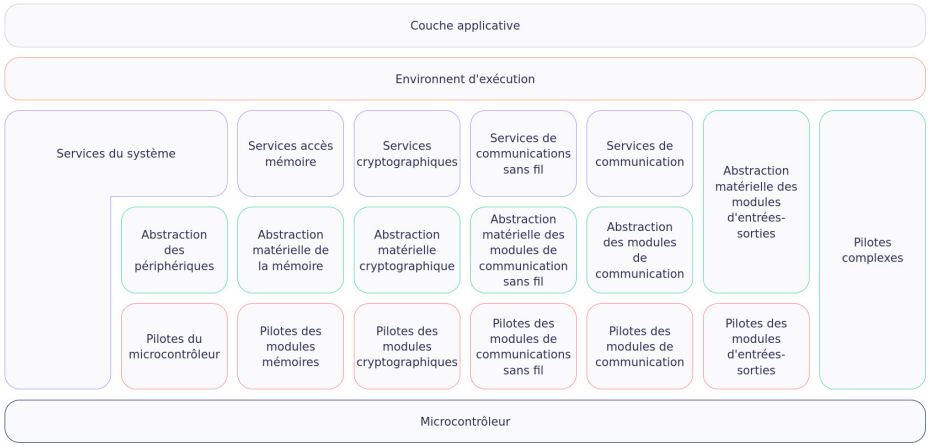

Fig. 8. Architecture logicielle de la plateforme *AUTomotive Open System ARchitecture* (AUTOSAR) Classic (adaptée de la documentation officielle [4]).

Cette architecture modulaire favorise une séparation claire entre les developpements, facilitant la maintenance, la mise à jour et la sécurité des logiciels embarqués. En particulier, la couche *services systèmes* intègre des mécanismes de sécurité tels que des services cryptographiques, la communication sécurisée entre ECUs, et des fonctions de diagnostic sécurisées, contribuant à la protection contre les accès non autorisés et les attaques potentielles.

En complément de cette isolation logique, des standards comme celui proposé par GlobalPlatform[8] [17] recommandent la mise en place d'une chaîne de confiance fondée sur une racine de confiance matérielle, généralement un composant de sécurité (SE). Cette chaîne permet de

[8] `https://globalplatform.org/automotive-initiative/`

démarrer un environnement d'exécution de confiance (TEE), sur lequel repose ensuite un environnement d'exécution riche (REE), tel qu'Android Auto [18]. Déployée initialement dans le domaine de la téléphonie mobile, cette architecture permet de garantir l'intégrité de l'environnement d'exécution tout en confiant les opérations sensibles à la TEE, bénéficiant ainsi à la fois d'une isolation forte et de performances adaptées aux besoins applicatifs. Néanmoins, cette architecture doit être adaptée aux contraintes strictes de sûreté de fonctionnement propres aux systèmes embarqués automobiles.

7 Conclusion

Dans cet article, nous avons présenté l'analyse de sécurité d'un véhicule représentatif du début des années 2020, et qui est encore en circulation. Ce véhicule cible embarque toutes les mesures de sécurité et de défense en profondeur attendues. De même, l'ensemble des mises à jour disponibles par le constructeur ont été appliquées. Ainsi l'analyse proposée s'applique à un véhicule à l'état de l'art de la sécurité embarquée, pour sa catégorie et sa génération.

Dans ce contexte, nous avons étudié la sécurité de l'implémentation de la pile Bluetooth. Ce protocole est en effet disponible dans la grande majorité des véhicules modernes au travers de l'infodivertissement. En particulier, nous avons exploité une vulnérabilité qui, d'après la `CVE-2018-20378` associée, n'impacte *pas* la version embarquée de l'implémentation cible. Néanmoins, nous avons été capable de revisiter en profondeur les mécanismes sous-jacents à cette vulnérabilité pour l'étendre au travers de nouvelles techniques d'exploitation qui sont détaillés dans cet article.

Les travaux présentés permettent de dérouter le flot de contrôle afin d'**exécuter un code arbitraire à distance** et **sans intervention des occupants du véhicule**, ce qui a été réalisé et démontré sur le véhicule cible. Dans les faits, cette exploitation permet à un attaquant de prendre le contrôle de l'infodivertissement et de forger dynamiquement des commandes CAN, ayant ainsi un impact direct sur le comportement routier et la sécurité du véhicule ciblé. Il reste important de noter toutefois que, comme précisé par le constructeur du véhicule étudié, les commandes CAN critiques sont filtrées en sortie d'ECU limitant ainsi les risques sur la sûreté des occupants et des usagers de la route. Une procédure de *responsible disclosure* a été réalisée pour permettre au constructeur de prendre les actions nécessaires en interne et auprès des prestataires concernés afin de corriger la vulnérabilité exposée.

Remerciements

Ce travail n'aurait pas été possible sans l'aide de nombreuses personnes qui, au détour de discussions enrichissantes, nous ont permis de faire progresser notre réflexion et d'avancer dans cette étude.

Nous tenons à remercier tout particulièrement Nicolas Godinho, Arnaud M., Alain Ozanne et Mathieu Renard pour leurs contributions indirectes mais significatives, qui ont permis de réaliser le travail présenté dans cet article.

Nous souhaitons également exprimer notre gratitude à Sébastien Varrette pour son soutien sans faille durant toute la réalisation de ce travail, ainsi que pour sa relecture attentive, qui a grandement contribué à l'amélioration de cet article.

Références

1. UN Regulation No. 155 : Uniform provisions concerning the approval of vehicles with regards to cybersecurity and cybersecurity management system. `https://unece.org/transport/documents/2021/03/standards/un-regulation-no-155-cyber-security-and-cyber-security`, mars 2021.
2. UN Regulation No. 156 : Uniform provisions concerning the approval of vehicles with regards to software update and software updates management system . `https://unece.org/transport/documents/2021/03/standards/un-regulation-no-156-software-update-and-software-update`, mars 2021.
3. Armis : Blueborne. `https://www.armis.com/research/blueborne/`.
4. AUTOSAR : Classic Platform architecture. `https://www.autosar.org/standards/classic-platform`, novembre 2024.
5. Ryad Benadjila, Mathieu Renard, José Lopes-Esteves et Chaouki Kasmi : One Car, Two Frames : Attacks on Hitag-2 Remote Keyless Entry Systems Revisited. *In 11th USENIX Workshop on Offensive Technologies (WOOT)*, Vancouver, BC, Canada, août 2017. USENIX Association.
6. Yulong Cao, S. Hrushikesh Bhupathiraju, Pirouz Naghavi, Takeshi Sugawara, Z. Morley Mao et Sara Rampazzi : You Can't See Me : Physical Removal Attacks on LiDAR-based Autonomous Vehicles Driving Frameworks. *In 32nd USENIX Security Symposium*, pages 2993–3010, Anaheim, CA, USA, août 2023. USENIX Association.
7. Barak Caspi : CVE-2018-20378. `https://nvd.nist.gov/vuln/detail/CVE-2018-20378`, mars 2019.
8. Core Specification Working Group : Bluetooth Core Specification, juillet 2021.
9. Gianpiero Costantino et Ilaria Matteucci : Reversing Kia Motors Head Unit to discover and exploit software vulnerabilities. *Journal of Computer Virology and Hacking Techniques*, 19(1):33–49, 2023.

10. CyberThreat Research Lab : Under Pressure : Exploring a Zero-Click RCE Vulnerability in Tesla's TPMS. https://vicone.com/blog/under-pressure-exploring-a-zero-click-rce-vulnerability-in-teslas-tpms, décembre 2024.
11. Sébastien Dudek, Jean-Christophe Delaunay et Vincent Fargues : V2G Injector : Whispering to cars and charging units through the Power-Line. *In Actes du Symposium sur la Sécurité des Technologies de l'Information et des Communications (SSTIC)*, Rennes, France, juin 2019. article (PDF).
12. Mikhail Evdokimov : 0-click RCE on the IVI component : Pwn2Own Automotive edition. *In Hexacon*, Paris, France, octobre 2024.
13. Mikhail Evdokimov et Radu Motspan : Remote Exploitation of Nissan Leaf : Controlling Critical Body Elements from the Internet. *In BlackHat Asia 2025*, Singapour, avril 2025.
14. Flavio D. Garcia, David Oswald, Timo Kasper et OswaPierre Pavlidès : Lock It and Still Lose It – on the (In)Security of Automotive Remote Keyless Entry Systems. *In 25th USENIX Security Symposium*, Austin, TX, USA, 2016. USENIX Association.
15. Ben Gardiner et Chris Poore : Trailer Shouting, Talking PLC4TRUCKS Remotely with an SDR. *In DEF CON*, août 2020. article (PDF).
16. Jessica Gerson : Hackers already infiltrate EV chargers. It could only get worse. https://grist.org/technology/hackers-already-infiltrate-ev-chargers-it-could-only-get-worse/, juillet 2022.
17. Globalplatform's Automotive Task Force : Cybersecurity in Automotive. https://globalplatform.org/resource-publication/cybersecurity-in-automotive-v1-0/, novembre 2022.
18. Google : Android Auto. https://www.android.com/intl/fr_fr/auto/.
19. Andy Greenberg : The Cheap Radio Hack That Disrupted Poland's Railway System. https://www.wired.com/story/poland-train-radio-stop-attack/, août 2023.
20. Thomas Huybrechts, Yon Vanommeslaeghe, Dries Blontrock, Gregory Van Barel et Peter Hellinckx : Automatic Reverse Engineering of CAN Bus Data Using Machine Learning Techniques. *In* Fatos Xhafa, Santi Caballé et Leonard Barolli, éditeurs : *Proceedings of the 12th International Conference on P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC)*, volume 13 de *Lecture Notes on Data Engineering and Communications Technologies*, pages 751–761. Springer, novembre 2017.
21. Koebler Jason : Why American Farmers Are Hacking Their Tractors With Ukrainian Firmware. https://www.vice.com/en/article/why-american-farmers-are-hacking-their-tractors-with-ukrainian-firmware/, 2017.
22. Muhammad Usman Khan, Naveed Ahmed, Fazal Rehman, Inayat Khan et Yousaf Muhammad : Cybersecurity in Vehicle-to-Grid (V2G) Systems : A Systematic Review. *arXiv preprint arXiv :2503.15730*, 2025.
23. Michael Kreil et Flüpke : Wir wissen wo dein Auto steht - Volksdaten von Volkswagen [*Nous savons où se trouve ta voiture - Données personnelles de Volkswagen*]. *In 38e Chaos Computer Club (CCC)*, décembre 2024.
24. Joe Kukura : Pranksters Claim They Can Stop Self-Driving Cars With Orange Cones , Waymo and Cruise Not Amused. https://sfist.com/2023/07/07/

pranksters-claim-they-can-stop-self-driving-cars-with-orange-cones-waymo-and-cruise-not-amused/, juillet 2023.

25. Charlie Miller et Chris Valasek : Remote Exploitation of an Unaltered Passenger Vehicle. *DEF CON*, août 2013.
26. Charlie Miller et Chris Valasek : Technical White Paper : Remote Exploitation of an Unaltered Passenger Vehicle. Rapport technique, IOActive, 8 2015. white paper (PDF).
27. Le Monde : L'autoroute qui recharge les véhicules électriques expérimentée dès 2025. https://www.lemonde.fr/economie/article/2024/09/23/l-autoroute-qui-recharge-les-vehicules-electriques-experimentee-des-2025_6328982_3234.html, septembre 2024.
28. Colin O'Flynn : BAM BAM!! On Reliability of EMFI for in-situ Automotive ECU Attacks. *In Proceedings os ESCAR conference*, 2020.
29. Jonathan Petit, Bas Stottelaar, Michael Feiri et Frank Kargl : Remote Attacks on Automated Vehicles Sensors : Experiments on Camera and LiDAR. *In BlackHat Europe*, Amsterdam, Pays-Bas, novembre 2015. article (PDF).
30. François Pollet et Nicolas Massaviol : Evolution et dé-évolution des systèmes multimédia embarqués. *In Actes du Symposium sur la Sécurité des Technologies de l'Information et des Communications (SSTIC)*, Rennes, France, juin 2016. article (PDF).
31. Redford, q3k et MrTick : Breaking "DRM" in Polish trains : Reverse engineering a train to analyze a suspicious malfunction. 37e Chaos Computer Club (CCC), décembre 2023.
32. Neiko Rivera, Sam Curry, Justin Rhinehart, Ian Carroll et Kenneth Lugo : Hacking Kia : Remotely Controlling Cars With Just a License Plate. https://samcurry.net/hacking-kia, septembre 2024.
33. SAE International : Power Line Carrier Communications for Commercial Vehicles. standard, novembre 2023.
34. Mohammad Ali Sayed, Ribal Atallah, Chadi Assi et Mourad Debbabi : Electric vehicle attack impact on power grid operation. *International Journal of Electrical Power & Energy Systems*, 137, 2022.
35. Ken Tindell : CAN Injection : keyless car theft. https://kentindell.github.io/2023/04/03/can-injection/, mars 2023.
36. W. van Beijnum et S. Laro-Tol : Hacking EV charging stations via the charging cable. *In Proc. of the Hardware Security Conference and Training (Hardwear.io NL 2024)*, Amsterdam, Pays-Bas, 10 2024.