

Développement d'un pare-feu domestique
Rapport de projet
Rémy Decocq

Année Académique 2018-2019
Master en Sciences Informatiques, bloc 1
Faculté des Sciences, Université de Mons

Table des matières

1 Présentation de l'<i>Internet des Objets</i>	5
1.1 Généralités	5
1.2 Caractéristiques des équipements de l' <i>IoT</i>	5
1.2.1 Domaines d'application	5
1.2.2 L'environnement <i>smarthome</i>	6
1.2.3 Restrictions des équipements	7
1.3 Protocoles adaptés à l' <i>IoT</i>	7
2 Présentations des pare-feux	9
2.1 Généralités	9
2.2 Différentes positions et fonctions dans l'architecture réseau	9
2.2.1 Les pare-feux niveau réseau	9
2.2.2 Les pare-feux niveau hôte	10
2.3 Les types de pare-feu	11
2.3.1 Pare-feu de filtrage (sans état)	11
2.3.2 Pare-feu à état	12
2.3.3 Pare-feu applicatif	12
2.3.4 Pare-feu applicatif proxy	13
2.3.5 Pare-feu identifiant	13
2.4 Systèmes de détection d'intrusion (<i>IDS</i>)	14
2.4.1 Les NIDSs	14
2.4.2 Les HIDSs	15
2.4.3 Les DIDSs	15
2.5 Systèmes de prévention d'intrusion (<i>IPS</i>)	16
2.6 Les pare-feux nouvelle génération	16
2.7 Les pare-feux de référence	17
2.7.1 Sous forme de software	17
2.7.2 Sous forme d'appliance	18
3 La sécurité dans l'<i>IoT</i>	20
3.1 Motivations et exemples	20
3.2 Description des besoins en terme de sécurité	21
3.3 Vulnérabilités liées aux équipements IoT	22
3.4 Architecture globale des systèmes impliquant des équipements IoT	23
3.5 Les archétypes d'attaques courantes	25
3.5.1 Sur la couche perception	25
3.5.2 Sur la couche réseau	25
3.5.3 Sur la couche support et application	26
3.6 Ressources, solutions et outils existants	27
3.6.1 Le travail d'OWASP	27
3.6.2 La standardisation de protocoles adaptés sécurisés	27
3.6.3 La détection des équipements en réseau et de leurs vulnérabilités	27
4 Développement d'une application <i>IoTMonitor</i> - introduction et concepts	30
4.1 Présentation générale	30
4.1.1 Contexte et environnement ciblé	30
4.1.2 Objectifs de l'outil	30
4.2 Composants du système de surveillance	32
4.2.1 Modules actifs et passifs - abstraction des programmes	32
4.2.2 Routine - automatisation de la surveillance	34
4.2.3 Netmap - recensement des équipements du réseau	35
4.2.4 Centre d'événements - logging et lancement d'alertes	36
4.3 Modèle d'abstraction des logiciels	36
4.3.1 Modèle général d'abstraction	36

4.3.2	Design pour un module actif	37
4.3.3	Design pour un module passif	38
5	Implémentation de l'application	39
5.1	Justification des choix généraux	39
5.2	Discussions des choix d'implémentation	40
5.2.1	Choix du langage	40
5.2.2	Gestion du temps	40
5.2.3	Exécution des programmes sous-jacents	40
5.2.4	Parsing des sorties	40
5.2.5	Stockage et agrégation des informations	41
5.2.6	Interactivité avec l'application	41
5.2.7	Alertes et notifications par email	42
6	Manuel d'utilisation	43
6.1	Prérequis et installation	43
6.2	Utilisation classique	43
6.2.1	Point d'entrée de l'application	43
6.2.2	Navigation dans les menus	44
6.2.3	Consultation de l'état des éléments	45
6.2.4	Exemple pratique d'utilisation	45
6.3	Utilisation du point de vue du programmeur	48
6.3.1	Écrire un module	48
6.3.2	Intégrer un module dans l'application	49
7	Tests et évaluation de l'application	50
7.1	Évaluation de la détection des équipements	50
7.2	Évaluation de la détection de menaces	50
7.2.1	Simulation d'équipements virtuels	50
7.2.2	Utilisation dans un réseau domestique peuplé	50
7.3	Déploiement sur Raspberry Pi et utilisation distante	50
8	Conclusion	51

Introduction

Depuis maintenant plusieurs années, la connectivité n'a cessé d'évoluer : en se limitant au domaine de l'Internet entre 2000 et 2015, une estimation de l'augmentation du pourcentage de la population mondiale l'utilisant avoisine 40% [5] [2]. Que ce soit dans le cadre d'infrastructures de type « mainframe » ou dans le contexte des ordinateurs personnels, les technologies et équipements relatifs au réseau et aux communications sont devenus indispensables. En raison du potentiel croissant d'interconnexion, les ordinateurs et équipements mis en réseau s'exposent à davantage de menaces. Heureusement, parallèlement à cette évolution, les performances de ces machines rejoignant de nouveaux réseaux ont également suivi une amélioration en terme de performances. Cela a permis d'en renforcer la sécurité à plusieurs niveaux et surtout d'intercepter efficacement les menaces étrangères liées à l'utilisation des réseaux. À l'heure actuelle, les OS utilisés classiquement sur des machines desktop fournissent un pare-feu simple (*Windows Defender*, un utilitaire fourni de base dans MAC OS X, *iptables/Netfilter* ou autre pour les distributions Linux). Ce dernier peut tourner en arrière plan de façon quasi imperceptible car il demande peu de ressources par rapport à ce qu'une machine actuelle peut offrir.

En parallèle avec la montée en puissance de ces machines de type desktop, serveurs, etc. s'est développée depuis à peu près les années 2000 la tendance de l'« Internet des Objets », ou encore plus communément abrégé IoT pour *Internet of Things*. Bien qu'assez large, cette dénomination regroupe beaucoup d'objets et de concepts, qu'ils soient virtuels ou non mais possédant un dénominateur commun : la capacité de communiquer en réseau avec d'autres équipements. Cela englobe par exemple la domotique, les outils et capteurs de mesures diverses, les imprimantes et scanneurs en réseau, etc. Tous ces éléments convergeraient idéalement vers une mise en réseau commune, leur permettant de communiquer malgré leur nature hétérogène [16]. Cela peut se faire via le réseau Internet, il n'est pas rare d'orienter ces connections vers un cloud permettant de traiter intelligemment la masse de données émises par ces équipements [20]. Or, comme évoqué ci-dessus, plus on s'interconnecte et plus on s'ouvre à des attaquants potentiels, ce qui pose problème si rien n'est mis en place pour s'en protéger.

Ce travail aura pour objectif premièrement de faire un état de l'art des dispositifs de protection qui sont actuellement déployés dans l'IoT et plus particulièrement dans le cadre domestique. Il s'agit d'un monde beaucoup plus hétérogène et restreint en terme de ressources que celui des ordinateurs desktop qu'on retrouve classiquement dans ce milieu. De fait il n'est pas toujours possible de réutiliser telles quelles toutes les technologies et ressources assurant la protection de ces machines non restreintes. De plus, il faut prendre en considération les spécificités d'une habitation : au centre de tous ces équipements communiquant se trouve l'habitant, une personne n'étant pas forcément apte à manipuler et contrôler ces nouvelles technologies. Il sera également question d'établir une vision globale des différents dispositifs de pare-feux existant et sous quelles formes ceux-ci peuvent être implémentés dans un réseau domestique. Deuxièmement, il sera question de mettre en pratique ces connaissances pour développer un système en lien avec la protection des réseaux domestiques et des équipements IoT qui les peuplent.

1 Présentation de l'*Internet des Objets*

1.1 Généralités

L'Internet des objets, qu'on désignera par *IoT* pour le terme plus répandu de *Internet of Things*, est une notion englobant un vaste ensemble de dispositifs. Aucune définition formelle n'est acceptée globalement, mais plusieurs organismes ont tenté d'en établir une ébauche. Par exemple, l'ITU-T (ITU Telecommunication Standardization Sector) Y.2060 le définit comme tel :

« *Global infrastructure for the society, enabling advanced services by inter-connecting (physical and virtual) things based on existing and evolving interoperable information and communication technologies.* »

Derrière cette définition très générale, on peut distinguer plusieurs sous-groupes d'objets connectés distincts, aux applications tant variées que hétérogènes, dans des domaines et secteurs également très différents. C'est ce qui fait la force et en même temps la faiblesse de cet ensemble d'équipements et services qu'on regroupe derrière le terme *IoT* et que des efforts considérables sont déployés pour inter-connecter au maximum. C'est un domaine d'étude intéressant car il représente littéralement ce qu'on pourrait considérer comme le futur de notre environnement technologique. De fait, les équipements que l'on peut associer à une partie de l'*IoT* ont déjà fait leur apparition dans notre quotidien : en 2017 on comptait 8,4 milliards de machines en présentant les caractéristiques et les estimations pour l'année 2020 tendent vers 20,4 milliards d'objets connectés d'après la société d'analyse Gartner [11].

1.2 Caractéristiques des équipements de l'*IoT*

1.2.1 Domaines d'application

Les secteurs dans lesquels l'*IoT* s'est implanté ces dernières années sont nombreux et très variés : ils s'étendent de l'industrie au domaine des soins de santé en passant par la tendance des *Smart Home*. C'est ce dernier domaine qui est approfondi dans ce travail et qui sera le plus sous-entendu par la suite quand le terme *IoT* est utilisé. La Figure 1 présente une vision d'un schéma global des autres domaines qui gravitent autour du vaste monde de l'*IoT*.

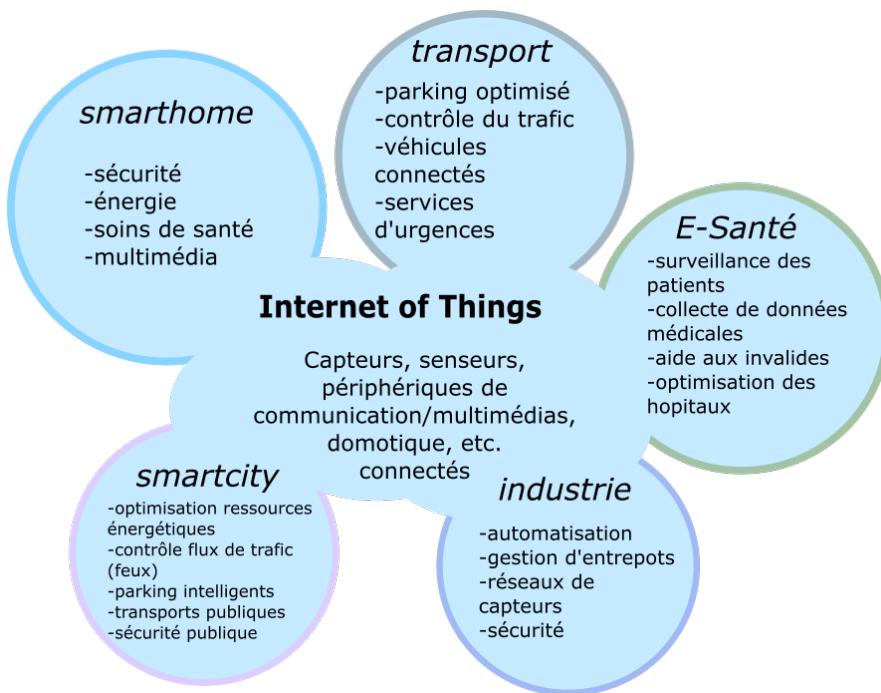


FIGURE 1 – Domaines d'application de l'*IoT*

1.2.2 L'environnement smarthome

Le terme émergeant *Smart Home/smarthome* est une fois de plus très englobant et général. Il n'en existe pas de définition formelle et communément acceptée. Basman M. Hasan et al. [9] en présentent plusieurs. Un résultat les unifiant pourrait être

« Une smarthome est un environnement lié au domicile particulier où plusieurs équipements ou sous-systèmes sont inter-connectés et où les informations qu'ils échangent sont collectées et utilisées afin de surveiller, réguler et automatiser l'écosystème du domicile ».

L'utilisateur en tant que personne physique y vivant est donc au centre de cette architecture, et y siège comme le principal intervenant. Effectivement, l'objectif global du déploiement de tous ces dispositifs est l'amélioration de sa qualité de vie. La notion d'intelligence est intrinsèquement liée avec celle de l'interconnexion de tous ces capteurs et actuateurs déployés dans l'environnement du domicile. Il s'agit d'en récolter et regrouper toutes les données en un point central doté d'une capacité de traitement plus évoluée afin qu'il puisse en tirer une optimisation globale de l'habitation (du système de sécurité, des économies d'énergie, de temps par l'automatisation, etc.).

Une certaine classification fonctionnelle peut être établie pour distinguer de façon plus concrète les différents équipements qui peuvent intervenir dans l'écosystème d'une smarthome. Elle est schématisée par la Figure 2, inspirée de [9]. Ce qui est désigné par *point d'interconnexion* peut dépendre de l'architecture réelle d'une smarthome [20]. Dans la plupart des cas il s'agit d'une machine faisant office de collecteur pour toutes les données transitant dans le domicile et de gateway vers le reste de l'Internet, éventuellement le cloud associé.

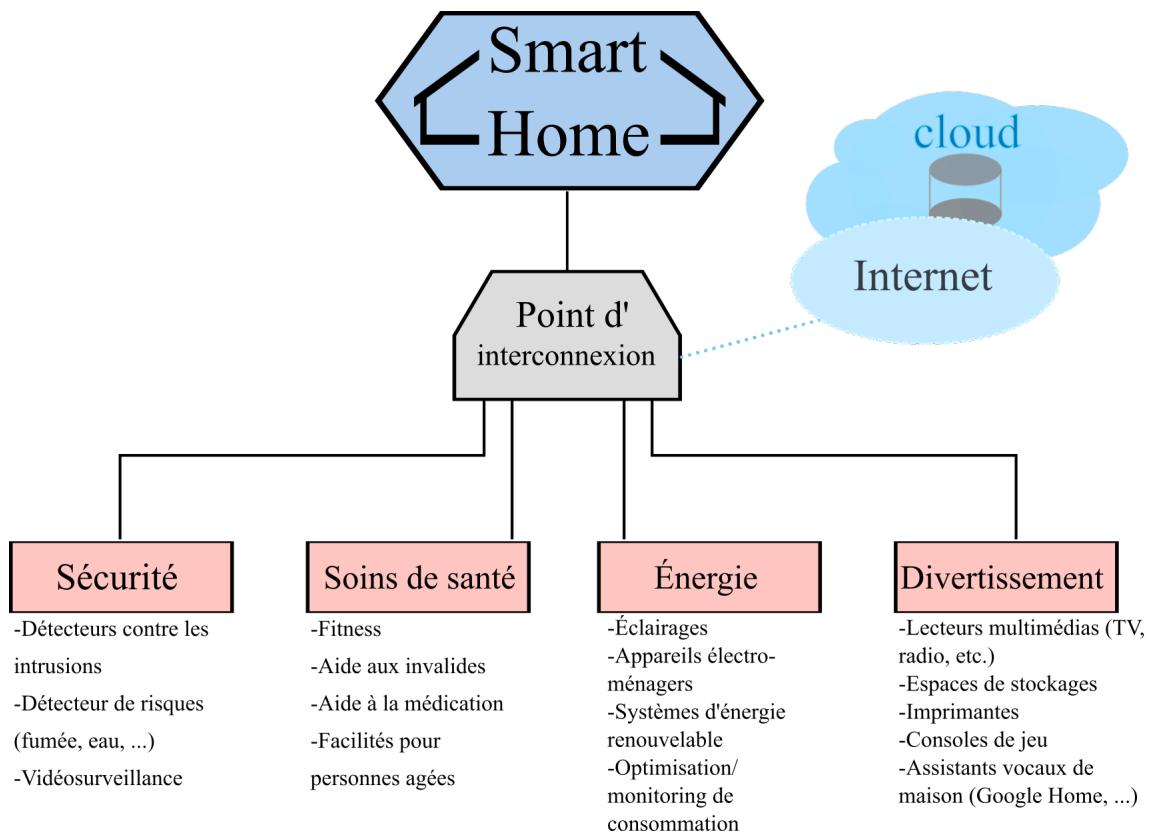


FIGURE 2 – Classification fonctionnelle des équipements IoT d'une smarthome

1.2.3 Restrictions des équipements

Malgré le fait que l'ensemble des objets considérés comme appartenant à l'IoT soit très hétérogène, on peut distinguer plusieurs caractéristiques communes à beaucoup d'entre eux. Elles tendent généralement vers ce qui est vu comme une restriction par rapport à un ordinateur type classique (*desktop*). Ces éléments constituent les plus gros freins au développement de la sécurité sur de tels système [8]. Les conséquences de ces restrictions sont discutées plus en détail dans la section 3 de ce document.

Conçus pour satisfaire une unique fonction

Le meilleur exemple est celui des capteurs. Un capteur a pour objectif de faire une mesure d'une grandeur physique (température, pression, etc.), d'en tirer une valeur numérique et de faire remonter via son interface avec le réseau cette information vers une unité centrale. Cette dernière accumule ainsi des mesures provenant des nœuds distribués pour y appliquer un traitement, et c'est à ce plus haut niveau que le processus de décision a lieu s'il est nécessaire. Ce genre d'équipement est généralement minimaliste au possible et ne peut donc pas remplir d'autre tâche.

Requièrent une faible consommation énergétique

Les systèmes embarqués n'ont pas toujours accès à une source illimitée d'énergie, et auront donc une durée de vie limitée à celle de leur batterie. En conséquence, il est souhaitable d'économiser un maximum, ce qui peut se faire en réduisant les temps d'éveil de l'équipement et en optimisant le nombre d'opérations effectuées quand il tourne à plein régime. Dès lors, certains protocoles et algorithmes doivent être adaptés (relatifs aux communications réseaux mais aussi à la sécurité) [20].

Sont contraints en ressources CPU, mémoire et radio

Ces contraintes sont aussi identiques à celles des systèmes embarqués classiques. En plus de celles-ci, on peut mettre en évidence le fait que les radios (quand il s'agit d'une interface sans fil) sont assez faibles et ne permettent pas des communications à haut débit. En résulte également que des techniques comme le saut de fréquence et les algorithmes de chiffrement de type asymétrique sont plus compliquées à mettre en œuvre [18].

Sont programmés à un bas niveau d'abstraction

Étant conçus pour ne remplir que des fonctions spécifiques, certains équipements ne sont pas programmables en utilisant des langages de haut niveau. Par conséquent, ce sont souvent des boîtes noires difficilement manipulables et statiques : les mises à jour et patch de sécurité ne sont pas déployables aisément par les constructeurs [20]. Les seules interactions que l'utilisateur lambda peut avoir avec l'équipement sont celles prévues par l'interface de ce dernier s'il y en a une (pour le configurer, consulter son état, etc.).

1.3 Protocoles adaptés à l'IoT

Les restrictions communément présentées par les équipements IoT énumérées dans la sous-section 1.2.3 imposent de repenser les protocoles utilisés dans un réseau non soumis à ces contraintes. Les protocoles décrits ci-dessous ont été conçus dans cette optique et sont utilisables dans un réseau composé d'équipements restreints en ressources, tel qu'un réseau de capteurs communiquant via la technologie sans-fil (WSN - *Wireless Sensors Network*). La figure 3 en présente un récapitulatif, mettant en exergue les différences avec les protocoles utilisés dans un réseau classique (supportant la pile *TCP/IP*).

Le standard **IEEE 802.15.4** a été défini par l'IEEE (Institute of Electrical and Electronics Engineers) pour les réseaux sans-fil dits « personnels » (WPANs - *Wireless Personal Area Networks*) c-à-d destinés à couvrir une zone d'émission de l'ordre de quelques dizaines de mètres. IEEE 802.15.4 supporte les couches physiques (PHY) et lien (MAC - *Medium Access Control*) dans ce cadre, en considérant le manque de ressources des équipements [13] [20].

Les réseaux utilisant IEEE 802.15.4 dans les couches plus basses sont contraints par la taille maximum que les trames peuvent contenir (pour les couches supérieures) selon ce standard, c-à-d 102 bytes. Sachant que la taille maximum d'un paquet IPv6 peut aller jusqu'à 1280 bytes par défaut, pour utiliser ce protocole il est nécessaire d'introduire un mécanisme d'adaptation. C'est l'objectif principal de la couche intermédiaire **6LoWPAN** : elle gère la fragmentation et le ré-assemblage des paquets IPv6 pour qu'ils puissent être transférés dans un réseau utilisant IEEE 802.15.4.

6LoWPAN propose d'autres services tels que la compression des en-têtes IPv6 (intuitivement nécessaire) et de l'auto-configuration de l'équipement joignant un réseau (similaire à IPv6).

L'aspect routage (sous-entendu pour des adresses IPv6) est généralement géré par le biais de **RPL** (bien qu'il soit possible de le faire avec 6LoWPAN). Il s'agit d'un protocole à vecteurs de distances adapté aux réseaux comme les WSN, comme le suggère son nom complet : *Routing Protocol for Low-Power and Lossy Networks*. Il se base sur une topologie virtuelle en arbre établie par un DODAG (*Destination Oriented Directed Acyclic Graph*), la racine étant un nœud qui a pour tâche de collecter toutes les informations montantes (et de les transmettre à un nœud extérieur au WSN). Plusieurs métriques peuvent être envisagées pour déterminer les routes, comme le nombre de sauts, la qualité des sources d'énergie de chaque nœud, les probabilités de réussite de transmission entre les nœuds. RPL est donc assez souple que pour s'adapter aux besoins et aux particularités des réseaux de natures variées que l'on peut retrouver dans l'IoT.

DTLS (*Datagram Transport Layer Security*) est le mécanisme de sécurisation sur lequel peuvent se reposer les applications utilisées par les équipements IoT. Son homologue dans les réseaux non restreints est TLS, qui se repose sur la fiabilité de transport garantie par TCP : si un paquet est pas ou mal délivré, la connexion est interrompue [17]. DTLS est conçu pour fonctionner par dessus **UDP**, qui est le protocole *de facto* utilisé pour le transport dans les réseaux IoT (celui-ci demandant moins de ressources). Il n'est pas forcément destiné à l'IoT, aussi des implémentations ont vu le jour dans cette optique ([TinyDTLS](#)).

Du côté applicatif, **CoAP** est un bon candidat. Ce protocole est maintenu par l'IETF, par l'équipe du CoRE (*Constrained RESTful Environments*) et est de fait basé sur l'architecture REST [13] comme son homologue HTTP. Il s'agit donc d'un modèle client-serveur. CoAP aspire à restreindre le dialecte HTTP à un sous-ensemble adapté aux contraintes des équipements IoT (aux communications 6LoWPAN) et est prévu pour fonctionner sur UDP. Ce dernier n'étant pas un protocole de transport fiable, CoAP implémente un mécanisme simple impliquant des messages *Confirmables* et *Acknowledge* pour pouvoir assurer un minimum de fiabilité si les données à transmettre sont critiques.

MQTT est un autre protocole qui diffère de CoAP sur plusieurs points¹. C'est un protocole de messagerie orienté souscription/publication léger par rapport à HTTP, la taille des messages se restreignant à quelques bytes contre plusieurs centaines [19]. L'architecture est également différente : on a un équipement dit *broker* qui joue le rôle de serveur de messagerie autour duquel s'articulent plusieurs clients désirant communiquer entre eux. Chaque client s'abonne à un *topic* pour lequel il est intéressé en établissant une liaison avec le broker, et recevra un message concernant ce topic posté par tout autre nœud. Cette liaison, ouverte en permanence, utilise TCP et peut donc être authentifiée et chiffrée avec TLS. En plus du fait que la longueur des noms des topics peut poser problème avec la taille maximum des frames 802.15.4, cette utilisation de TCP pose problème pour les nœuds trop limités en ressources. Un port vers UDP du nom de **MQTT-SN** a été établi à cette fin.

Couches du modèle	<i>Protocoles de la stack IoT</i>	<i>Protocoles de la stack TCP/IP</i>
Application	IETF CoAP MQTT ...	HTTP FTP DNS ...
Sécurité	DTLS	TLS
Transport	UDP	TCP UDP
Réseau	IPv6 IETF RPL	IPv4 IPv6
Adaptation	IETF 6LoWPAN	Non défini
Lien	IEEE 802.15.4 MAC	Dépend du média et de la technologie (IEEE 802.11 WiFi, Ethernet, ...)
Physique	IEEE 802.15.4 PHY	

FIGURE 3 – Comparaison du modèle en couche classique (TCP/IP) avec celui adapté à l'IoT

1. https://www.eclipse.org/community/eclipse_newsletter/2014/february/article2.php

2 Présentations des pare-feux

2.1 Généralités

Un pare-feu est un dispositif, virtuel ou matériel, qui surveille et contrôle le lien entre un réseau dit de confiance et un réseau extérieur non fiable. Typiquement, ce réseau potentiellement dangereux est Internet et la zone à protéger est le réseau interne d'une entreprise ou d'une habitation. L'existence des pare-feux est une conséquence du besoin d'outils de protection aux bordures de réseaux distincts contrôlés par des entités différentes. D'une part, il faut garantir que des données internes confidentielles restent à l'intérieur du réseau de confiance. D'autre part, il est nécessaire de filtrer les données entrantes et sortantes de ce réseau de sorte qu'aucune menace ne s'y infiltre par des flux, même initiés depuis l'intérieur du réseau de confiance. Ces filtres sont créés et assemblés à partir de *politiques* ou *règles* définies par défaut ou par les personnes compétentes liées à l'entité gérant le réseau.

2.2 Différentes positions et fonctions dans l'architecture réseau

Deux classes de pare-feux sont distinguables en fonction de ce qu'ils visent à protéger. La première correspond aux pare-feux au niveau réseau (*network firewalls*), situés en bordure de LANs, WANs et intranets. Ceux-ci, de par leur nature de barrière entre réseaux, peuvent également fournir des services plus évolués : système de NAT, gestion de *zones démilitarisées* (définies ci-dessous), service DHCP, etc. [14] La seconde classe agit au niveau des noeuds du réseau eux-mêmes (*host-based firewalls*), protégeant une machine physique et non pas un réseau entier. Ces pare-feux se présentent donc sous forme de logiciels, directement intégrés au niveau du système d'exploitation ou installés à un plus haut niveau.

La figure figure 6 schématise la place de ces dispositifs, mettant en scène le scénario simplifié d'un utilisateur surfant sur le site d'une entreprise. Une requête HTTP pour une page est créée et envoyée depuis le domicile, destinée aux serveurs de l'entreprise. Ceux-ci sont protégés derrière une machine dédiée uniquement à la fonction de pare-feu (qu'on qualifie alors par le terme *appliance*), dans une zone tampon désignée par **DMZ** pour *Demilitarized Zone* (zone démilitarisée). Cette zone intermédiaire est un sous-réseau contenant les machines étant destinées à être accédées depuis Internet mais qui n'ont aucun besoin d'accéder au réseau local de l'entreprise (typiquement des serveurs). Ainsi, si une attaque contre ces machines aboutit et que l'attaquant en prend le contrôle, il n'a pas encore un accès direct aux autres machines de l'entreprise (si le pare-feu est toujours fonctionnel). Cela permet en plus d'établir un filtrage plus fin effectué par le pare-feu, qui utilise des règles différentes en fonction de la zone ciblée et donc des services à considérer comme légitimes.

2.2.1 Les pare-feux niveau réseau

Afin de remplir leur fonction, ces pare-feux sont placés en bordure du réseau à protéger et sont directement liés aux machines qui font office de *gateway* (routeurs). Le trafic échangé entre les réseaux ainsi connectés passe donc par le pare-feu afin d'être analysé et filtré, ce qui peut représenter beaucoup de données à traiter. Le pare-feu doit offrir une vitesse de traitement proportionnelle aux débits et à la qualité des liens qui le traversent afin de ne pas être un goulot d'étranglement. C'est pourquoi ces pare-feux doivent être très efficaces et sont communément portés (partiellement) au niveau matériel. On parle de *hardware-based firewall appliances*, qui sont des machines physiques dont le seul objectif est de remplir les tâches d'un pare-feu le plus efficacement possible. On y retrouve deux composants : la partie applicative software ou firmware remplissant la fonction de pare-feu, qui repose sur la deuxième partie plus basse constituée de juste ce qu'il faut d'un OS particulier (*jeOS - just enough Operating System*). Cet OS est généralement propriétaire, lié au fabricant du hardware sur lequel les deux parties opèrent et donc optimisé pour garantir les performances requises.

À plus petite échelle, dans un réseau domestique par exemple, on retrouve également des pare-feux directement implantés dans le routeur qui fait office de *gateway* pour l'habitation. Ceux-ci sont fatalement moins efficaces et complets que les matériels spécialement dédiés à cette unique fonction.

À titre indicatif, la Figure 4 donne un exemple d'une *appliance* pare-feu issue de la gamme des Cisco ASA 55xx-X, la dernière génération que Cisco a introduit sur le marché en 2018 [3]. Il ne s'agit pas du modèle le plus performant de la gamme en comparant les [performances et fonctionnalités](#), mais convient typiquement bien pour être déployé dans une petite entreprise (son prix avoisine 500€). Dans le cas d'une habitation, l'analogue correspondant est dans la majorité des cas le modem distribué par le fournisseur d'accès internet au domicile. La société Proximus fournit par exemple des *b-box*, que leurs clients peuvent configurer par une interface web. Dans ces configurations, il existe bien un onglet intitulé *firewall*, mais celui-ci ne présente que peu d'options de configuration comme le montre la Figure 5. Dans d'autres onglets, on peut trouver d'autres fonctionnalités qui peuvent y être relatives : restriction d'accès pour certains noeuds en fonction de l'heure (par adresse MAC) et blocage de certains sites par domaine. Cela semble assez faible, les utilisateurs du réseau ont tout intérêt à installer sur leurs équipements personnels des pare-feux pour hôtes.

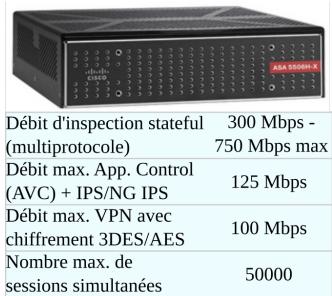


FIGURE 4 – Cisco ASA 5506H-X

Level	Low	Medium	High	?
Low: All connection attempts coming from the WAN or initiated from the LAN are permitted				
Medium:	All connection attempts coming from the WAN are rejected with the exception of those that have been authorized through port forwarding, DMZ or remote access configuration.			
All Services initiated from the LAN are permitted.				
High:	All connection attempts coming from the WAN are rejected with the exception of those that have been authorized through port forwarding, DMZ or remote access configuration.			
Services initiated from the LAN are restricted to the ones authorized via the rules set in the firewall (common services are covered by default).				

FIGURE 5 – Options de config. du pare-feu pour une b-box 3V

2.2.2 Les pare-feux niveau hôte

Du fait que les pare-feux de cette classe sont généralement déployés sur des machines de type desktop, la dénomination *pare-feu personnel* est aussi utilisée. Ces ordinateurs sont généralement munis de tels pare-feux par défaut ou peuvent tout du moins supporter leur installation par l'utilisateur si l'OS utilisé est classique (Windows, MAC OS X, ...). Le pare-feu agit jusqu'au niveau applicatif [15] et sous forme d'un *service* ou d'un *daemon* en fonction du système d'exploitation utilisé. Cela permet une proximité étroite avec l'OS et les autres processus. Le pare-feu personnel est donc capable de contrôler le trafic réseau demandé par chaque application et de détecter les menaces engendrées par certains flux. Ces dernières peuvent être entrantes ou sortantes : une machine extérieure qui tente d'établir une connexion suspecte ou un exécutable sur la machine à protéger qui initie un trafic avec une cible blacklisted, par exemple.

Les principales fonctionnalités d'un pare-feu personnel devraient être au minimum les suivantes [14]

- Bloquer les attaques et comportements dangereux du réseau extérieur : scanning des ports ouverts, attaques par fragmentation, *IP Spoofing*, etc.
- Stopper les menaces venant de l'intérieur : un exécutable comme un *malware* ou un *spyware* qui tente d'établir une connexion vers l'extérieur doit être bloqué et mis en quarantaine
- Présenter de l'automatisation car d'une part car un utilisateur non-expérimenté pour sa configuration doit quand même rester protégé et d'autre part il doit pouvoir se mettre à jour automatiquement
- Agir au niveau applicatif : un malware pourrait utiliser le port web 80 pour se répandre par exemple, pour détecter cela il faut analyser les payloads des paquets et disposer d'une base de données à jour pour y déceler des patterns malicieux sur base de leur signature
- Alerter l'utilisateur quand un événement survient, et le logger avec suffisamment d'informations pour qu'il puisse prendre une décision adaptée
- Éviter les *faux-positifs* (bloquer du trafic légitime)

Que ce soit en entreprise ou dans une habitation, il est donc probable que l'on retrouve des pare-feux appartenant à ces deux classes distinctes là où ils sont efficaces (voir Figure 6). Il ne s'agit que d'échelles différentes, qui impliquent également des intervenants différents. En entreprise, l'administrateur sécurité/réseau configue du matériel spécifique afin de sécuriser les frontières de son réseau tout en préservant ses performances. Il est aussi possible qu'il installe dans les machines internes de son réseau des pare-feux assez puissants pour empêcher les propagations des attaques déclenchées. Dans le domicile, l'habitant est sommairement protégé par le routeur fourni par son FAI qui inclut un pare-feu réseau par défaut. S'il est un minimum expérimenté, il sera à même d'installer et configurer des pare-feux personnels sur chacun de ses équipements connectés, pour autant que cela soit possible (par exemple, les contraintes liées aux équipements de l'IoT décrites dans la sous-section 1.2.3 sont un frein à cela).

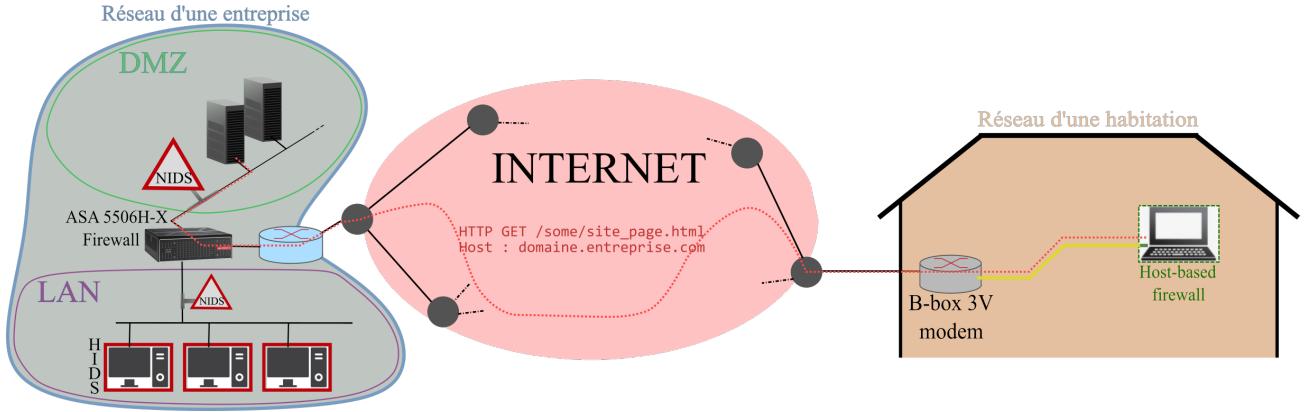


FIGURE 6 – Schéma récapitulatif des places des différents dispositifs de protection dans un réseau

2.3 Les types de pare-feu

2.3.1 Pare-feu de filtrage (sans état)

Aussi désignés dans la littérature par le terme *stateless*, ces pare-feux sont les plus rudimentaires, le premier prototype étant celui élaboré par Jeffery Mogul en 1989 [24]. Généralement, ces pare-feux sont rapides mais peu efficaces car facilement dupés. Un pare-feu de ce type inspecte chaque paquet individuellement et détermine s'il peut passer sur base d'un ensemble de règles écrites que ses en-têtes vérifient ou non. Ces règles portent sur les en-têtes TCP/IP (également UDP), les options y attenant et l'interface d'entrée. Plus en détail : les adresses IP source et destination, les ports source et destination au minimum seront soumis aux filtres du pare-feu. Il y a plusieurs problèmes avec cette approche :

- Il n'y a aucune vérification au dessus de la couche transport : la partie applicative peut contenir n'importe quoi
- Le pare-feu n'est pas dynamique : il n'apprend rien du trafic qu'il laisse passer. Aucun état n'est retenu, alors que par exemple TCP est un protocole lié à une machine à état (orienté connexion) dont il pourrait être possible de garder une trace
- Les règles sont très redondantes à écrire pour être efficaces, très peu modulables et facilement sujettes aux erreurs et oubli

Les paquets ICMP peuvent également faire l'objet d'un filtrage du même acabit [14]. Ce type de pare-feu tombe en désuétude car il est trop simple à duper par des attaquants.

2.3.2 Pare-feu à état

Le but de ce type de pare-feu dit *stateful* est d'améliorer les techniques de filtrage sans état en maintenant de l'information sur les flux passant au travers, principalement TCP. Effectivement, au contraire d'UDP, TCP est un protocole qui garde un état qui se traduit dans certains champs de ses headers. Dès lors, le pare-feu peut accéder à cette information, l'analyser et l'utiliser pour en déduire dans quel état est actuellement la connexion entre les deux processus communiquant sur les deux hôtes impliqués. Cela peut se traduire par l'inspection des flags (SYN, FIN, ...), des numéros de séquences et acquittements qui permettent au pare-feu de contrôler en conséquence le contenu d'une table interne des connexions.

L'établissement d'une connexion TCP se fait comme suit : lors de la réception du premier paquet TCP par le pare-feu, celui-ci va le soumettre à ses règles de filtrage définies comme dans un pare-feu sans état. Si le paquet est valide, il est retransmis pour poursuivre librement sa route et une entrée est ajoutée dans la table interne du pare-feu. Celle-ci va changer d'état à chaque paquet de la communication reçu, jusqu'à arriver dans un état dit *established* correspondant à la fin du 3-way handshake TCP. Tous les paquets suivants pourront alors être traités rapidement en établissant la correspondance entre ses en-têtes et celles de l'entrée en état established dans la table (qui sont donc théoriquement également légitimes). Cette façon de procéder est plus efficace que de soumettre chaque paquet individuellement aux règles de filtrage qui peuvent être complexes et demander beaucoup de travail et donc gourmandes en temps.

Dans le cas d'une communication utilisant UDP, le contrôle possible est moins fin car il ne s'agit pas d'un protocole à état et les échanges ne sont pas bidirectionnels. Dans ce cas, lorsque le premier paquet d'une transmission arrive en entrée du pare-feu, il est soumis aux règles de filtrage. S'il est considéré comme correct, le pare-feu ajoute une entrée dans sa table pour les couples d'adresses et ports et considérera les suivants comme légitimes. Cette entrée expirera après un temps donné sans autre nouveau paquet la matchant.

2.3.3 Pare-feu applicatif

Ce type de pare-feu peut être considéré comme une extension complète aux simples pare-feux à états dans le but d'en améliorer la fiabilité. Là où ces derniers sont capables de déterminer quels protocoles sont utilisés et autorisés sur chaque port, les filtres ajoutés au niveau applicatif peuvent en plus déduire à quelles fins sont utilisés ces protocoles (l'inspection va jusqu'à la couche 7 du modèle OSI). Il s'agit cependant de plus que de simples filtres, c'est une technologie à part entière qui est utilisée : l'inspection profonde des paquets ou *deep packet inspection* (DPI). Les pare-feux utilisant la DPI mêlent les fonctionnalités d'un pare-feu à état avec les systèmes de détection et prévention d'intrusions (sous-sections 2.4 et 2.5). La décision de bloquer ou de laisser passer le paquet tient compte de ce qu'il contient réellement au niveau données et de l'interprétation qui en est faite pour déterminer s'il représente un danger.

Par exemple, les *Web Application Firewalls* (WAF) sont une sous-catégorie des pare-feux applicatifs qui opèrent un filtrage sur le trafic HTTP. Ils peuvent donc filtrer le trafic sur base de certaines règles d'accessibilités fixées en bloquant des requêtes aux URL douteuses ou considérées comme non éthiques vis-à-vis de l'organisme maintenant le réseau par lequel elles transitent. Outre cela, les WAFs permettent également de se protéger contre les attaques dites de type *parameter tampering* (injection SQL, cross-site scripting, etc.) [23], ainsi que d'inclure des filtres anti-spam pour les courriels [7].

Plus généralement, les pare-feux applicatifs permettent de se protéger des virus, vers et tentatives d'*exploits* de faiblesses connues des systèmes à protéger [1] [23]. Un autre avantage d'analyser jusqu'à la couche 7 grâce à la DPI est que certains protocoles (FTP par exemple) utilisent la couche applicative pour transmettre des informations relatives aux couches plus basses (adresses IPs, ports, etc.) et qui seront impliquées dans des communications passant par le dispositif pare-feu [7]. Comme illustré par l'exemple des WAFs, le principal défaut inhérent à leur nature réside dans le fait que pour chaque type de trafic applicatif que l'on souhaite contrôler, un travail spécifique doit être opéré (établir des règles dédiées par exemple) [4].

2.3.4 Pare-feu applicatif proxy

Ces pare-feux mêlent l'inspection au niveau applicatif et le rôle de proxy pour les flux qui sont destinés à le traverser originellement. Ainsi, il devient réellement un intermédiaire entre les deux parties communiquant, interceptant les paquets pour les analyser puis les retransmettant vers le destinataire comme s'il en était la source [14]. Avant cette retransmission, le paquet peut être inspecté du point de vue de toutes ses couches et déterminé comme étant dangereux en fonction des opérations de filtrages effectuées dans le proxy. Un contrôle très fin peut donc être opéré. Cela garantit un niveau de sécurité plus haut que les pare-feux de filtrage pur qui traitent les paquets à la volée [6].

Cette approche possède cependant trois gros désavantages :

- Clairement, l'impact sur les performances du réseau n'est pas insignifiant (surtout si le pare-feu n'est basé que sur du software pur et pas sur une machine adaptée)
- À chaque nouveau protocole applicatif que l'on souhaite pouvoir filtrer de la sorte, un nouveau dispositif proxy correspondant doit être pensé et développé pour en inspecter le trafic
- Si un proxy présente une faille de sécurité, un assaillant peut l'utiliser comme un vecteur d'attaque et prendre le contrôle du système sur lequel il tourne. S'il parvient alors à désactiver les services relatifs au pare-feu, la voie vers ce qu'on souhaitait protéger se retrouve toute ouverte

2.3.5 Pare-feu identifiant

Pour ce type de dispositif, on veut pouvoir définir des règles de filtrage en fonction de l'utilisateur ou le groupe qui se cache derrière un paquet ou un flux. Plusieurs schémas d'association existent, définissant ce à quoi la notion d'un utilisateur correspond. Par exemple certains de ces pare-feux comme `authpf`² sous OpenBSD, qui étant placés sur un gateway, exigent qu'avant d'autoriser un flux l'utilisateur d'un hôte communiquant doit établir une connexion SSH avec ce dernier. Un autre type d'association possible peut se faire par adresse MAC.

Les pare-feux destinés aux entreprises développés par Cisco (ASA) [12] et Palo Alto Networks [15] utilisent des gestionnaires d'annuaires d'utilisateurs basés sur les services `Windows Active Directory`. Dans ces bases de données utilisateurs, on retrouve les différentes adresses IP associées à un utilisateur particulier ainsi que les groupes qu'ils composent. Les interactions avec le pare-feu sont transparentes à ce niveau, en effet ce dernier travaille avec les objets « utilisateurs » et non plus directement les IPs (bien que cela reste possible). Procéder de cette façon apporte plusieurs avantages [12] :

- Cela simplifie la création et la gestion des politiques de sécurité du pare-feu
- Offre la possibilité d'identifier facilement les utilisateurs utilisant le réseau
- Simplifie la surveillance des activités des utilisateurs du réseau et permet d'identifier la source de menaces

2. <http://www.openbsd.org/faq/pf/authpf.html>

2.4 Systèmes de détection d'intrusion (*IDS*)

Ce type de systèmes, abrégé par *IDS* pour *intrusion detection system*, est analogue à ce qu'est un système d'alarme intérieur dans l'environnement d'une habitation : quand les dispositifs de protection mis en place à l'entrée sont contournés (une présence non permise est détectée à l'intérieur), l'alarme est lancée et des actions sont éventuellement prises en conséquence. Les *IDSs* sont donc des sentinelles qui surveillent le réseau interne, logiquement placées après le pare-feu dans le sens entrant. Ces systèmes se présentent sous la forme d'outils spécialisés dans l'interprétation des logs des routeurs, pare-feux, serveurs et autres agents du réseau interne. Les *IDSs* sont épaulés par une base de données des signatures d'attaques déjà connues et y comparent le contenu des logs afin de trouver des *patterns* qui correspondent [14]. Dans une telle situation, plusieurs actions allant de simplement alerter l'administrateur réseau à couper les accès réseaux des machines peuvent être déclenchées, en fonction du degré de certitude (correspondance forte entre les signatures) et de la menace.

Tout comme les pare-feux, les *IDSs* peuvent être des ressources software ou reposent sur du hardware spécifique. Dans le cas où il s'agit de software, ils sont établis sur la même machine que le pare-feu, les proxy ou autres dispositifs de bordure de réseau. S'ils se présentent sous forme d'équipements hardware spécifiques, ils sont installés de sorte à contrôler et surveiller de près un de ces dispositifs sensibles. Le trafic tant entrant que sortant peut être analysé par les *IDSs*, car les attaques peuvent autant venir de l'extérieur que se déployer depuis l'intérieur (chevaux de Troie, spywares, etc.).

L'analyse des événements repose communément sur deux techniques. La première, la détection par signature (*signature detection*), utilise une base de données de signatures d'attaques déjà connues et se base sur le trafic et les patterns observés pour établir un match avec ces signatures. La seconde approche est celle de la détection d'anomalies (*anomaly detection*). Il est question d'utiliser des heuristiques afin de distinguer les situations et comportements anormaux, sur base de profils types construits par analyses statistiques, agencements de règles ou réseaux neuronaux. Outre la technique utilisée, trois catégories d'*IDSs* existent en fonction de leur place dans l'architecture réseau (leur mode de fonctionnement diffère en conséquence) :

- Surveillance du réseau (*NIDS* : *Network-based IDS*)
- Surveillance des systèmes du réseau (*HIDS* : *Host-based IDS*)
- Surveillance distribuée du réseau (*NIDS* : *Distributed IDS*)

2.4.1 Les NIDSs

Ces dispositifs surveillent le réseau ou un segment de ce réseau, sous forme d'un équipement intermédiaire de capture (ses interfaces sont en mode écoute pour l'entièreté du trafic transitant par le segment). Il est important d'avoir plusieurs unités de surveillance distinctes dans le cas où le réseau est scindé en plusieurs modules/zones. Par exemple, un serveur web d'une entreprise pourrait être infecté et servir de plateforme de lancement d'une attaque depuis le réseau interne, vers un autre module contenant des serveurs internes.

Deux manières de procéder à l'analyse sont possibles : soit en mode *in-line* soit en mode *off-line*. Le cas de la capture du trafic par interface décrit ci-dessus correspond au mode *in-line*. Le trafic est analysé en temps réel, ce qui permet une plus grande réactivité mais peut constituer un goulet d'étranglement si le processus de décision est coûteux. Le mode *off-line* est donc plus avantageux à ce niveau, puisque le processus d'analyse et décision est opéré sur des données stockées et non pas à traiter à la volée. Une inspection plus fine est alors envisageable, cependant le principal défaut de ce mode de fonctionnement reste le manque de réactivité à une attaque.

2.4.2 Les HIDSs

Les IDSs orientés systèmes, *Host-based IDS*, sont élaborés dans le but de protéger uniquement l'hôte sur lequel ils sont déployés et non plus un segment du réseau. Ils ciblent également de façon précise le trafic qu'ils doivent surveiller en fonction de la nature du système. Par exemple, si l'hôte ne maintient aucun service DNS, il est inutile d'analyser des requêtes DNS qui lui parviendraient pour y déceler une menace exploitant une faille connue dans le protocole DNS. Puisque les HIDSs vont s'exécuter sous forme d'un processus (*daemon*) sur la machine hôte qui elle-même fonctionne sous un OS classique, un lien doit être établi entre les deux pour que l'HIDS puisse surveiller le système et les interfaces réseaux. Certains vont même jusqu'à rechercher les intrusions dans le noyau de l'OS.

D'une part, un HIDS peut surveiller le comportement du système dynamiquement en récupérant des informations que ce dernier met à sa disposition à la manière d'un antivirus. Différents indicateurs sont à interpréter :

- Activité de la machine même : processus qui y vivent, ressources qu'ils consomment (CPU, RAM, réseau etc.), modification dans les comptes utilisateurs
- Activité des utilisateurs : commandes entrées, programmes lancés, tentatives d'accès à des ressources non autorisées, passage au compte administrateur
- Patterns d'exécution ou de procédure de déploiement des vers, virus, chevaux de Troie (shell ouvert simultanément à l'ouverture d'un fichier, accès anormaux aux interfaces réseau, ...)

D'autre part, certaines sections plus critiques du système peuvent être intéressantes à analyser finement, car c'est généralement là qu'un attaquant voulant prendre le contrôle de la machine va laisser des traces en y installant ses outils softwares. Un HIDS voulant s'assurer que des sections critiques (système de fichiers, registres, ...) ne sont pas infectées va maintenir une base de données de leur évolution dans le temps, sous forme d'attributs et sommes de contrôles (*checksums*). Ces dernières permettent d'assurer l'intégrité des ressources concernées en en comparant les valeurs régulièrement dans le temps, et notifiant les différences observées.

2.4.3 Les DIDSs

Les DIDSs se présentent sous la forme d'une architecture distribuée d'IDSs (surveillant des segments de réseaux et/ou hôtes), et d'une unité centrale de management qui récolte toutes les informations de ces dispositifs déployés. Ainsi, la machine centrale peut maintenir une vaste base de données centralisée, représentant l'état global du réseau à protéger. Cette approche présente plusieurs avantages :

- moins de faux positifs car plus de données pour justifier une prise de décision (agrégation des événements et attaques)
- mises à jour et distribution de la base de données des signatures aisées
- centralisation des alertes/logs, contrôle global avec une vue d'ensemble
- réponses aux événements plus efficaces, IDSs plus simples à administrer en conséquence (ajout de nouvelles règles suite à des brèches découvertes, blacklisting d'IPs, etc.)

Afin de transmettre les alertes générées jusqu'à l'unité centrale efficacement, les différents nœuds IDSs du réseau devraient communiquer de façon homogène. À cette fin, un standard a été écrit sous le nom de format IDMEF (*Intrusion Detection Message Exchange Format*, RFC 4765). Ce langage d'alertes est lisible par l'humain, utilisant le format XML. Utilisé par tous les nœuds qui effectuent la tâche de détection d'intrusion (quelque soit le type et l'implémentation), l'unité centrale qui récolte toutes les alertes peut utiliser un outil de pilotage de la sécurité du réseau global comme [Prelude](#)³. Cet outil normalise, trie, agrège, corrèle et en tire les conclusions sur des décisions à prendre pour protéger le réseau, tout en fournissant une interface (et des logs) via laquelle l'administrateur peut surveiller ce qui se passe globalement.

3. [https://en.wikipedia.org/wiki/Prelude_SIEM_\(Intrusion_Detection_System\)](https://en.wikipedia.org/wiki/Prelude_SIEM_(Intrusion_Detection_System))

2.5 Systèmes de prévention d'intrusion (IPS)

Dans la section précédente, il était question de détection des attaques et pour certaines d'entre elles des actions à effectuer en contre-mesure pour bloquer la menace. Cela relève du domaine d'application des systèmes de prévention d'intrusion, aussi abrégé IPS pour *intrusion prevention systems*. Tout comme pour les IDSs, deux types d'IPS peuvent être distingués en fonction de leur place dans l'architecture réseau : les *host-based IPS* agissent au niveau local pour un hôte, tandis que les *network-based IPS* défendent un réseau ou un de ses segments internes. De fait, IDSs et IPSs vont de paire puisque les contre-mesures prises par un dispositif IPS sont basées sur ce que l'IDS détecte et lui transmet comme information.

En plus de lancer une alerte à destination d'un administrateur et écrire des logs complets, les actions qu'un IPS peut effectuer suite à la détection d'une menace sont généralement les suivantes :

- Jeter les paquets détectés comme contribuant à cette attaque
- Bloquer tout trafic issu de la même adresse IP/du même utilisateur que celui considéré comme attaquant (mettre fin à la connexion TCP impliquée)
- Reconfigurer le pare-feu associé pour qu'il puisse à l'avenir bloquer le trafic relatif à cette attaque
- Si l'IPS est lié à un dispositif faisant du proxy, il peut agir sur le contenu des paquets pour neutraliser la menace (par exemple ôter un fichier infecté joint à un mail)

2.6 Les pare-feux nouvelle génération

Abrégés par NGFW (*next-generation firewalls*), ces pare-feux font suite à ce qu'on distingue comme la troisième génération dans l'évolution des dispositifs pare-feux [4], c'est-à-dire ceux qui appliquent les concepts de filtrage au niveau applicatif décrits dans la sous-section 2.3.3. Les NGFW effectuent une analyse plus fine que ces derniers en utilisant les technologies avancées de *deep packet inspection*. Les services qu'offrent la nouvelle génération de pare-feux s'étendent encore au-delà de la simple fonctionnalité d'inspection des paquets, et sont devenus des systèmes très complexes implémentant d'autres dispositifs liés à la sécurité dont les suivants :

- Support des technologies de (dé)chiffrement utilisées classiquement dans les réseaux
- Les systèmes de management et contrôle sur base des identités des utilisateurs inspirés des pare-feux identifiants
- Les pare-feux destinés aux flux web (WAFs), le filtrage d'URLs selon des politiques de sécurité et éthique
- Des moyens de contre-mesures aux attaques empruntés aux IPSs
- Les environnements virtuels (*sandboxing*) de tests pour déceler dans un flux suspect une attaque non reconnue [15]
- De vastes bases de données sur le cloud interrogées efficacement via un service spécialisé fourni avec le pare-feu (environnement WildFire et similaires)

2.7 Les pare-feux de référence

2.7.1 Sous forme de software

Ces pare-feux sont destinés à être installés et utilisés par-dessus des systèmes d'exploitation génériques tels que Windows, distributions Linux et BSD, etc. En plus de protéger la machine sur laquelle ils opèrent des menaces extérieures, certains d'entre eux intègrent d'autres fonctionnalités telles que du *traffic shaping* ou la gestion d'un système de NAT.

Windows

Windows Firewall est intégré par défaut avec les versions de l'OS Windows postérieures à *Windows XP*. Il s'agit d'un pare-feu qui se veut « user-friendly » : par défaut il présente une liste de profils pré-configurés pour que l'utilisateur puisse sélectionner celui qui convient le plus à ses besoins (*Public*, *Private* et *Domain*). Il agit comme un pare-feu à état basique mais assez complet dans sa fonction restreinte cependant, en plus d'être simple à utiliser et de disposer d'une interface graphique. Le fait qu'il soit directement incorporé dans le système permet un contrôle fin de quelles applications sont autorisées pour quels types de trafics.

Linux

Netfilter est la solution de base de filtrage de paquets inclue dans le noyau Linux. Plusieurs modules liés au noyau sont également fournis pour servir d'intermédiaire dans la configuration de Netfilter, notamment sous la forme de tables de règles en chaînes à appliquer séquentiellement sur les entrées à filtrer. Ces modules sont accessibles depuis l'utilitaire *iptables*, sous les noms de *ip_tables*, *ip6_tables*, etc.

nftables a pour objectif de remplacer Netfilter à terme car il est sensé apporter plus de performances principalement. L'utilitaire permettant de faire le lien avec cette nouvelle solution implémentée dans le noyau est réduit à *nft* (réduction du nombre de module par rapport à Netfilter).

Shorewall repose sur Netfilter, faisant office de couche d'abstraction pour faciliter la configuration des règles qui est assez laborieuse avec les outils comme *iptables*. Il permet l'écriture des règles alimentant les tables par un mécanisme de fichiers de configuration. C'est un software complet, il propose des fonctionnalités classiques telles que du NAT, du *traffic shaping*, un support VPN et **autres**. Une autre fonctionnalité notable est la possibilité de partitionner le réseau en zones distinctes (ou pouvant se recouvrir l'une l'autre), permettant le contrôle de toutes les connexions entre chaque paire de zone à considérer. Cela permet notamment l'établissement aisément de zones démilitarisées (DMZ).

BSD

Packet Filter (pf) est comparable à Netfilter pour les noyaux Linux. C'est une solution de filtrage de paquets à état, incluant d'autres fonctionnalités classiques telles que la gestion de NAT, des mécanismes liés à la *qualité de service* (QoS), une gestion des logs avancée en offrant leur configuration règle par règle. D'autres utilitaires sont venus s'y greffer, tel que *authpf* qui permet un système de gateway authentifié (voir les pare-feux identifiants). Packet Filter a fait l'objet de port vers de nombreux autres systèmes d'exploitation, notamment Apple Mac OS X et iOS, NetBSD et Solaris.

2.7.2 Sous forme d'appliance

Le terme *appliance* désigne un appareil/système constitué du strict minimum de composants pour remplir au mieux une fonction spécifique. Du point de vue des dispositifs pare-feux, on peut distinguer trois catégories d'appliances :

- les pare-feux destinés aux systèmes embarqués, particulièrement contraints en ressources comme c'est le cas pour certains équipements de l'IoT.
- les pare-feux logiciels destinés à fonctionner au dessus d'une couche d'abstraction du hardware spécifique (ou même sur une machine virtuelle), basés sur un noyau d'OS qui est transparent à l'utilisation
- les pare-feux reposant sur du hardware (machines dédiées vendues telles quelles) et qui utilisent le matériel et les équipements spécifiques construits à cette fin apportant de meilleures performances

Généralement, il s'agit de solutions destinées aux entreprises de par leur prix et leur nature. Effectivement, comme ce sont des équipements/dispositifs dédiés (presque) uniquement à la fonction de pare-feu (et à la sécurisation du réseau en général), ils sont souvent déployés comme des pare-feux niveau réseau. À noter que beaucoup de ces pare-feux tournent sur un OS propriétaire de l'entreprise qui les fabrique (Cisco, Check Point, etc.). Cela permet aux fabricants d'optimiser leurs produits tant au niveau performance que sécurité, restreignant leur OS au strict nécessaire (certains d'entre eux sont basés sur des noyaux déjà existants).

Appliances sur systèmes embarqués

IPFire est dérivé du noyau Linux et est capable de remplir toutes les fonctions basiques d'un routeur utilisé en conjugaison avec un pare-feu, sous des contraintes de ressource de 1GHz pour le CPU, 1GB de RAM et 4GB d'espace disque. De fait, IPFire peut être utilisé sur des systèmes comme une carte Raspberry PI ou dans des environnements virtuels. IPFire est modulable avec des [add-ons](#) gérés par un utilitaire de management, *Pakfire*. Cependant, [la version minimaliste](#) supporte déjà des fonctionnalités telles que l'inspection des paquets à état, le proxying, la prévention d'intrusions, la segmentation en zones du réseau, la gestion de QoS, les services DHCP et DNS, etc.

IPCop est un pare-feu open-source assez minimalistique utilisant Netfilter en arrière-plan. Les pré-requis pour le faire tourner sont : 200MHz de fréquence CPU, 64 Mo de RAM et 800 Mo d'espace de stockage. Même avec cette configuration minimale, il assure les fonctionnalités de proxy et d'IPS en plus du filtrage à état classique. Des modules peuvent être ajoutés en fonction des services supplémentaires requis : services DHCP ou DNS, *traffic shaping*, liste d'accès utilisateurs, etc. IPCop est une branche divergente du projet [SmoothWall](#), ce dernier s'étant développé de telle sorte qu'il n'est plus associable à un pare-feu *embarqué* en plus d'être passé en partie sous licence commerciale.

Appliances orientées software non spécifiques aux équipements restreints

pfSense se repose sur FreeBSD (bien qu'aucune connaissance sur ce dernier ne soit requise, puisque pfSense est une appliance à utiliser tel quel). Le software peut être installé sur une machine physique dédiée ou dans un environnement virtuel. Il est complètement open-source, et [à la prétention](#) d'offrir tous les services que les grandes distributions incluent dans leur appliances physiques en un seul software, gratuit et extensible. pfSense opère comme un routeur et pare-feu sur la machine dédiée, ces services étant alors configurables et mis à jour via une interface web.

Untangle est quant à lui basé sur le noyau Linux. Il est également déployable sur une machine simple dédiée ou virtuelle. Ses [fonctionnalités](#) incluent entre autres les classiques d'un pare-feu nouvelle génération (filtrage du contenu, IPS, identification utilisateur, etc.) mais aussi la possibilité de faire du cache web, du *load balancing* et du routage. Cependant, seule une version légère du software est distribuée gratuitement.

Softwares reposant sur des appliances hardware

CheckPoint Software est une entreprise qui vend des machines dédiées à la protection des réseaux ainsi que les solutions logicielles qui les accompagnent. Le software *GAiA* fait office d'OS pour les [appliances hardwares vendues par CheckPoint](#) pour lesquels il est optimisé, mais peut également être déployé sur une appliance virtuelle compatible hébergée sur une [machine qui en est capable](#).

Cisco offre des services similaires avec ses pare-feux nouvelle génération ASA 5500-X. Le software Cisco *Adaptative Security Appliance* est l'OS dédié principalement aux machines de la famille ASA (encore une fois des solutions de virtualisation existent). La sur-couche *Cisco Firepower Threat Defence* (FTD) est un software intégrant les fonctionnalités d'ASA et des services axés IDS/IPS nouvelle génération de [FirePOWER](#).

Ces solutions sont généralement utilisées dans des réseaux de grande taille, donc en entreprise. Les appliances hardwares sont vendues telles quelles à prix élevé, et le software fourni avec y est intégré pour en garantir les performances, tant au niveau des débits que de la détection et du blocage des menaces. Cependant, il n'est pas vain de s'y intéresser car certaines technologies sur lesquelles elles sont basées sont parfois réutilisables à plus petite échelle (dans le cadre de solutions pour protéger un réseau domestique entre autres). Par exemple, FirePOWER a été développé à partir du software open-source [Snort](#) et utilise les bases de données de définitions de signatures virales issues d'un utilitaire anti-virus nommé [ClamWin](#), libre également.

3 La sécurité dans l'IoT

3.1 Motivations et exemples

À priori, dans le cadre d'une smarthome, les équipements relevant de l'IoT sont plutôt des petits accessoires, voire gadgets (montres, lampes, balances connectées par exemple). On pourrait donc se dire qu'introduire des mécanismes de sécurité avancés dans de tels petits systèmes n'est pas primordial, l'aspect fonctionnel étant plus mis à l'honneur. C'est d'ailleurs ce qu'ont fait beaucoup de constructeurs, mettant sur le marché des équipements possédant des failles de sécurité importantes ou des faiblesses au niveau du design de conception. Un exemple interpellant est le fait que pour un équipement qui possède une interface accessible par un portail de sécurité par identification (requérant une entrée de la forme *user/password*), les valeurs d'usine par défaut sont laissées telles quelles, l'utilisateur n'étant jamais invité à les modifier [19]. Ainsi, il existe des listes établies de ces identifiants par défaut en fonction du constructeur (pour les webcams par exemple) qui, combinées avec des outils puissants comme Shodan ou Censys, constituent des brèches dans la protection de l'habitation pour les attaquants qui savent les exploiter. La Figure 7 illustre la déconcertante facilité avec laquelle il est possible d'attenter à la vie privée des utilisateurs inconscients (une unique requête a été nécessaire).

IP Webcam
86.205.251.170
amuseille-552-1-18-170.wf6-205.abo.wanadoo.fr
Orange
Added on 2018-12-16 20:32:02 GMT
France, Mâlemort
Technologie: swf B



```
HTTP/1.1 200 OK
Connection: close
Server: IP Webcam Server 0.3
Cache-Control: no-store, no-cache, must-revalidate, pre-check=0, post-check=0, max-age=0
Pragma: no-cache
Expires: -1
Access-Control-Allow-Origin: *
Content-Type: text/html
```

IP Webcam
107.181.14.128
Phoenix Cable
Added on 2018-11-26 10:05:11 GMT
United States, Phoenix City
Technologie: swf B F



```
HTTP/1.1 200 OK
Connection: close
Server: IP Webcam Server 0.4
Cache-Control: no-store, no-cache, must-revalidate, pre-check=0, post-check=0, max-age=0
Pragma: no-cache
Expires: -1
Access-Control-Allow-Origin: *
Content-Type: text/html
```

InsideCamera



The image is regularly refreshed every 30 seconds.

Pan / Tilt Scan
Preset Program
1 2 3 4
5 6 7 8
Brightness STD
Resolution 640x480
320x240
160x120
Image Quality Favor Clarity
Standard
Favor Motion
Image Size x1.0 x1.5
Buffered Image Start Capture Viewer
Multi-Camera Top Page Help

Android webcam server

If no video appear below, you are using unsupported browser

Tested browsers: Mozilla Firefox, Google Chrome

Open camera controls



Turn on LED Turn off LED

Autofocus Cancel autofocus

Take full-res picture Autofocus and take full-res picture

Audio, in case your external player does not support separate streams:

Click here to play audio with browser

Click [here](#) to play audio in external media player. Right click [here](#) and select "save as" to record it.

[Fullscreen view](#) (click video to change aspect behavior, F11 to remove browser borders).

FIGURE 7 – Résultats d'une brève recherche avec Shodan ciblant des caméras en réseau non ou peu protégées

20

Évidemment, l'utilisateur « final » n'est pas le seul qui puisse en subir les conséquences. Ses équipements peuvent entre autres rejoindre un ensemble d'autres objets connectés infectés, reconfigurés malicieusement en vue d'effectuer une attaque de masse de type *DDOS* sur une cible étrangère au réseau domestique. C'est ce qui s'est passé avec *Mirai* [27], un malware catégorisé comme *botnet* décelé bien après son expansion et qui a été utilisé dans plusieurs attaques menées sur le fournisseur de services DNS *Dyn*, la société d'hébergement *OVH* et bien d'autres. Mirai n'est pas un cas isolé, et les hackers ont pleinement saisi l'opportunité qu'incarne l'IoT : il existe beaucoup de failles connues et exploitables sur des équipements hétérogènes, et celles-ci sont en partie rendues publiques sur par exemple des dépôts gitbooks et des forums dédiés.

Les équipements qu'on retrouve dans une smarthome ne sont pas les seuls à présenter des lacunes en terme de sécurité. Encore plus grave, les systèmes connectés utilisés dans l'industrie de nos jours peuvent être attaqués, et cela peut porter lourdement à conséquence (jusqu'à l'échelle internationale). Le *scénario Horus* [25] le démontre particulièrement bien. Imaginée par un chercheur néerlandais, cette attaque cible les installations photo-voltaïques qui fournissent en énergie l'Europe, plus spécifiquement les onduleurs connectés y opérant (qui, brièvement, sont les dispositifs transformant le courant alternatif en courant continu). Pas moins de 17 failles ont été découvertes, dont certaines permettant de prendre le contrôle total des onduleurs à distance. Il serait alors possible pour un attaquant de simuler artificiellement l'impact qu'une éclipse solaire aurait sur le réseau, qui est en temps normal compensé artificiellement (celle-ci pouvant être anticipée). En appliquant une telle variation de puissance soudaine, le réseau entrerait dans un état de fluctuation tel qu'il ne serait plus contrôlable et une grande partie du réseau finirait par sauter. La perte suite à une telle panne, considérant qu'elle dure trois heures, a été estimée à 4,5 milliards d'euros.

3.2 Description des besoins en terme de sécurité

Ces notions sont similaires à celles généralement mises en avant dans le domaine de la sécurité informatique en général. On les agrémentera en faisant le lien avec le cas un peu plus restreint des équipements relatifs à l'IoT, si possible. Ces grands axes sont les suivants [18] [20] :

Confidentialité

L'accès aux informations doit être limité aux personnes et systèmes qui y sont autorisés, et uniquement eux : aucun accès indésirable ne doit être possible. Dans un réseau constitué de plusieurs équipements, les communications entre chacun des nœuds doivent être protégées des intrus pouvant potentiellement les intercepter.

Intégrité

Les informations stockées et échangées ne doivent pas être altérées, quelle qu'en soit la raison (de source malveillante ou non). En plus des informations, le firmware exploitant les équipements doit également respecter cette propriété (le système ne doit pas être compromis).

Disponibilité et continuité

Les systèmes doivent être fonctionnels quand cela est attendu et garantir l'accès et la bonne exécution des services qu'ils sont destinés à offrir, dans le délai attendu. La sécurité doit pouvoir être assurée en cas de panne ou dysfonctionnement de certaines parties du système, et ce même dans le cas de systèmes dont les ressources sont limitées (équipements sur batterie en fin de vie par exemple).

Authentification

Seuls les utilisateurs légitimes (acquéreurs du produit entre autres) devraient être en mesure d'accéder aux systèmes et de les configurer, ainsi que d'accéder aux informations sensibles qu'ils possèdent. L'authentification peut aussi être relative à une entité, comme le fabricant certifié du produit qui tente d'atteindre le système pour effectuer sa mise-à-jour ou lui appliquer un patch afin de corriger une faille dans sa sécurité.

3.3 Vulnérabilités liées aux équipements IoT

Les caractéristiques généralement présentées par les équipements décrites comme des restrictions dans la sous-section 1.2.3 portent à conséquence en terme de sécurité. Il en résulte de potentielles vulnérabilités [20], couvertes ou non par ces équipements en fonction de leur conception et implémentation spécifique. Celles-ci sont brièvement décrites ci-dessous.

L'exemple des caméras IP présenté en introduction l'illustre bien : l'**accessibilité aux équipements depuis le réseau** (surtout Internet) est un problème conséquent, dans la mesure où il n'est pas pris en considération à la conception du système. Des mesures découlant du principe d'authentification doivent être prises afin de limiter les accès aux entités légitimes uniquement.

Une autre vulnérabilité découle directement du fait que les équipements sont **restreints en ressources CPU et mémoire**. On y retrouve des petits microcontrôleurs travaillant sur 8 bits [20], ce qui contraint beaucoup l'utilisation des algorithmes de chiffrement classiques (AES, 3DES, TEA, etc.). C'est donc une vulnérabilité importante qu'il est capital de contrer car transmettre des données en clair va à l'encontre des principes de confidentialité énoncés plus tôt. Des solutions sous forme d'algorithmes adaptés [10] ou de matériel spécialement dédié au chiffrement⁴ existent.

Outre les restrictions en terme de capacité, une autre composante est à prendre en compte : **la source d'énergie limitée** (pour les équipements utilisant une batterie par exemple). Cette vulnérabilité ouvre la porte aux attaques d'exhaustion, consistant à désactiver un équipement par le biais de l'épuisement de sa source d'alimentation [21] (attaque dite de *Depletion-of-Battery*). C'est une atteinte au principe de disponibilité.

Les **firmwares non mis à jour** et globalement les **constructeurs peu impliqués dans la sécurité de leurs produits** constituent des vulnérabilités qui peuvent être exploitées par des attaquants. Par exemple, les identifiants par défaut ou stockés en clair. Peu d'équipements installés dans les smarthomes bénéficient d'un service de mise à jour permettant des patchs de sécurité [20] suite à des failles décelées et rendues accessibles publiquement.

L'**accès physique au(x) média(s) de transmission** représente une autre vulnérabilité, surtout dans le cadre des communications sans-fil. Dans l'environnement d'une smarthome par exemple, un dispositif de sécurité installé pour protéger l'habitation qui utilise du sans-fil devrait tenir compte du fait que toutes les communications peuvent être entendues en dehors de l'enceinte des murs de la maison (par des *outils de sniffing*). En plus de l'écoute passive, le média peut aussi être pris d'assaut activement, dans l'exemple précédent une attaque de type *Denial Of Service* pourrait impacter l'efficacité du système de sécurité.

L'**hétérogénéité des équipements** peut être problématique pour d'une part définir des standards et d'autre part avoir une bonne connaissance technique de chacun des équipements évoluant par exemple dans une smarthome. La documentation pour chaque équipement devrait alors lui être spécifique, et pour peu qu'elle existe il faudrait qu'elle développe les mécanismes de sécurité fournis avec l'équipement. Allant de paire avec cela, les systèmes de mises-à-jours et patchs de sécurité divergent de l'un à l'autre.

Dans le cas particulier des smarthomes, le **manque de connaissances des utilisateurs** en terme de sécurité informatique [22] peut être considéré comme une vulnérabilité car il peut en résulter une mauvaise installation ou configuration des systèmes menant à des failles exploitables. Conjugué avec le fait que les constructeurs des équipements que l'habitant achète ne mettent pas la priorité sur leur sécurisation, cela ouvre de multiples portes que les attaquants n'ont qu'à ouvrir pour attenter à la vie privée des utilisateurs.

4. <https://www.zyngbit.com/make-things-secure-1/>

3.4 Architecture globale des systèmes impliquant des équipements IoT

Leloglu propose dans [18] un travail de synthèse de plusieurs sources dont il dérive entre autres un modèle structurel d'une architecture typique de l'IoT. Celui-ci s'applique à un réseau domestique tel que présenté au point 1.2.2, dont on suppose certains équipements travaillent avec un cloud (communications extérieures au réseau local). C'est une représentation en couches, une couche supérieure s'appuyant sur les services fournis par les couches plus basses afin de remplir la fonction qui lui est dédiée. On distingue les couches suivantes (reprises par des exemples dans le tableau 1 et schématisées par la figure 8) :

La couche perception

Il s'agit de la couche la plus basse, où on considère le rôle des équipements IoT par rapport à l'environnement dans lequel ils évoluent. Comme expliqué dans la sous-section 1.2.3, ces derniers sont généralement conçus pour remplir une fonction spécifique. C'est dans cette couche qu'elle prend tout son sens, car elle dépend souvent de l'environnement. L'exemple le plus évident est les capteurs, possédant ce qui est nécessaire en terme de matériel pour effectuer une mesure d'une grandeur physique. Les divers systèmes embarqués peuvent également s'intéresser à leur environnement extérieur ou au(x) dispositif(s) qu'ils supervisent (lave-vaisselle connecté, etc.). Une autre technologie que l'on situe à cette couche est celle de la radio-identification (RFID - *radio frequency identification*), considérée par certain comme les prémisses à l'IoT [11]. La perception se fait ici au niveau du lecteur RFID, qui capte et interprète les tags RFID des objets qui évoluent dans son environnement. En résumé, l'objectif de cette couche est l'identification des objets uniques et la collecte d'informations de l'environnement physique.

La couche réseau

Cette couche a pour objectif le transfert des informations récoltées par la couche perception depuis les équipements qui les ont réalisées et jusqu'au système qui en effectuera le traitement. On distingue généralement deux parties dans ce transfert, la première correspondant à l'acheminement des données à travers un réseau local jusqu'à un portail (le gateway). Celui-ci qui constituera le point de départ de la deuxième partie du transfert, utilisant un réseau non spécifique tel que Internet pour atteindre le système de traitement terminal. Afin de rendre possible les communications entre ces deux réseaux de natures différentes, il est parfois nécessaire d'introduire une couche d'adaptation dans la pile des protocoles utilisés, e.g. *6LoWPAN* [26] qui permet des échanges basés sur IPv6 dans les réseaux contraints en ressources.

La couche support

La couche support englobe tout les systèmes de traitement des données, celles-ci ayant été acheminées jusqu'au centre de traitement par le réseau. Il s'agit donc de partir des données brutes transmises et de leur appliquer un processus de transformation. Ce dernier peut comprendre de l'agrégation de données, de l'analyse, du tri, de la classification, etc., l'objectif étant d'en tirer des données utiles et pertinentes. Les données finales peuvent ainsi être stockées dans une base de données sur laquelle une ou des applications peuvent travailler.

La couche application

Les applications sont diverses et variées, développées dans des domaines tels que repris dans la figure 1. Pour remplir leurs objectifs, elles utilisent les données obtenues en sortie du processus de traitement opéré par la couche inférieure (les couches support et application étant très proches, on les considère parfois comme une seule).

Niveau dans le système	Exemples de technologies, systèmes et équipements y opérant
Couche perception	Caméras, microphones, capteurs, GPS et divers systèmes embarqués mesurant et interagissant avec l'environnement qui leur est extérieur (montres intelligentes, détecteurs, etc.), ainsi que la technologie RFID. Côté software : OS adapté aux systèmes embarqués tels que Contiki, tinyOS, LiteOS, etc.
Couche réseau	Réseaux de capteurs, réseau mobile 2G/3G, réseaux classiques tel que Internet (IP).
Couche support	Cloud, centre de traitement des données, base de données et autres technologies et supports permettant d'agrégier, stocker et analyser les données (<i>Big Data</i>)
Couche application	Applications relatives aux smarthomes (interfaçage avec l'utilisateur, optimisation de son quotidien, etc.), applications médicales (surveillance de patients, aides aux invalides, etc.), sociales et structurelles (smartcity : parkings intelligents, etc.), scientifiques (études des mesures, détection de catastrophes, etc.) et industrielles (arrosage de plants, transport, etc.)

TABLE 1 – Architecture d'un système IoT proposée par Leloglu [18]

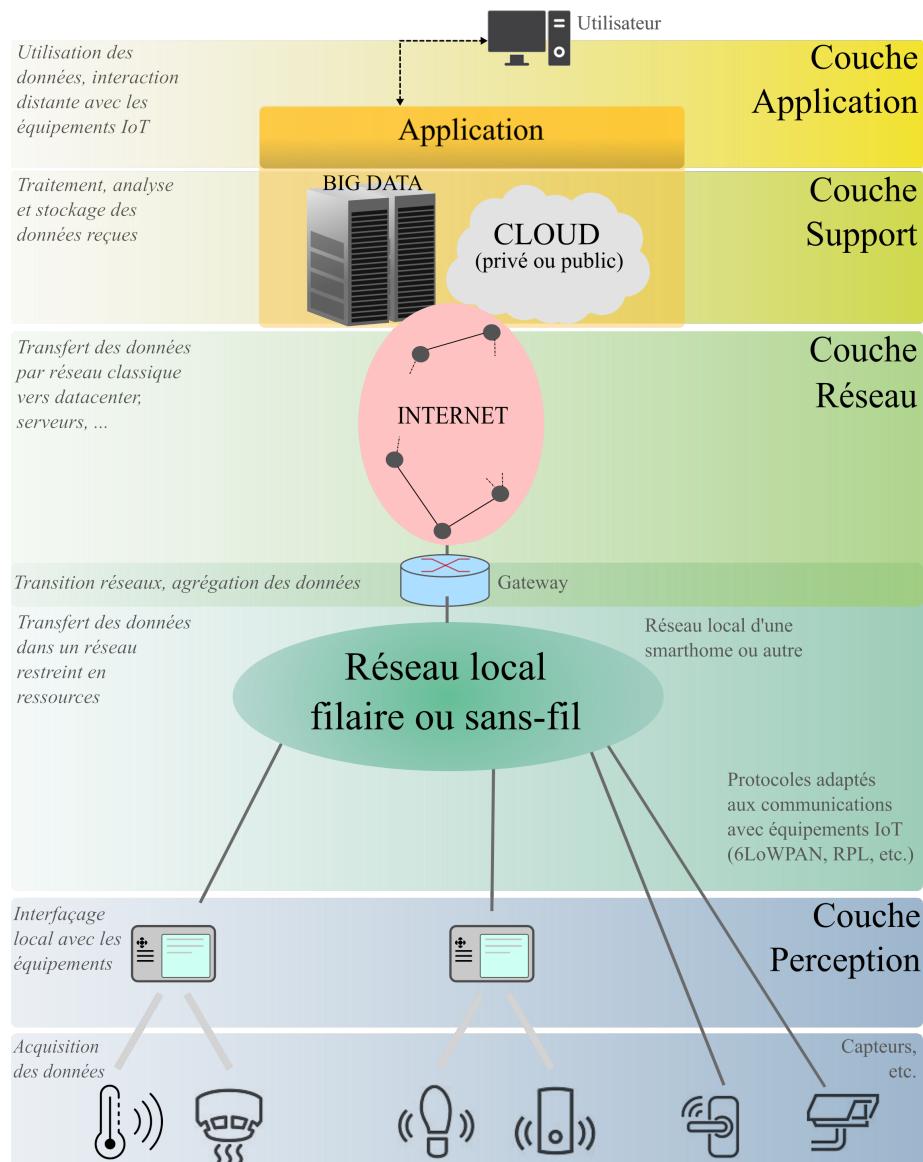


FIGURE 8 – Schématisation des couches considérées pour un système utilisant des équipements IoT

3.5 Les archétypes d'attaques courantes

Plusieurs schémas d'attaques peuvent être distingués, exploitant ou non des vulnérabilités recensées dans la section 3.3. Ces scénarios d'attaques typiques sont classés en fonction de la couche dans laquelle ils opèrent, selon Makhdoom et al. [21] et Leloglu [18].

3.5.1 Sur la couche perception

Drainage de batterie (*Depletion-of-Battery attack*) : en apparence justifiée car il s'agit d'un scénario légitime du point de vue de l'utilisation, ce type d'attaque porte préjudice à un nœud en le bombardant de requêtes qu'il doit traiter car considérées comme légitimes. Cela empêche le nœud de passer en mode veille pour préserver de l'énergie, et peut mener à l'extinction de celui-ci une fois sa source d'énergie épuisée.

Sabotage du nœud (*Node capturing/cloning*) : un attaquant peut attenter physiquement au système : il est possible qu'il accède à un nœud parmi ceux dispersés dans un réseau de capteur par exemple. Il lui est donc potentiellement possible de le corrompre ou le remplacer physiquement par un nœud intrus qui effectuera un travail dissimulé.

Brouillage radio (*Signal/Radio jamming*) : un type d'attaque DoS qui vise à occuper le canal (sans-fil) au moment où des nœuds souhaitent également émettre, brouillant les communications.

Écoute clandestine (*Eavesdropping*) : dans le cadre des communications sans-fil, un attaquant peut placer physiquement ou corrompre un nœud alors clandestin qui lui permet d'écouter les informations passant par ce canal. Ces informations peuvent être sensibles, telles que des mots de passe transmis en clair par d'autres équipements. Outre cela, le simple fait de savoir quand des informations sont émises sur le canal peut profiter à un attaquant, par exemple en l'a aidant à évaluer quand une maison est occupée ou non.

Attaque par intrusion ((*Semi-)**Invasise intrusions*)⁵ : par l'étude des variations de certains indicateurs tels que le voltage, le signal de l'horloge, etc., un attaquant peut inférer les opérations qu'un microcontrôleur effectue. Il est aussi possible de déduire certaines valeurs stockées dans la mémoire sensée rester privée telles que les clés maîtres de chiffrement (AES ou autre). En appliquant des fluctuations physiques, il est aussi possible de faire en sorte que le CPU exécute au final une autre opération que celle attendue.

Mises à jour faussées : des implémentations mal faites des mécanismes d'authentification peuvent aider un attaquant à soit maintenir une version du firmware obsolète, soit faire passer une mise à jour falsifiée comme provenant du constructeur officiel de l'équipement. Les patchs relatifs à la sécurité peuvent ainsi être bloqués, et les failles qu'ils corrigeaient laissées exploitable.

3.5.2 Sur la couche réseau

Spoofing : effectuée par le biais de paquets forgés, ce type d'attaque vise à usurper une identité renseignée comme source du paquet falsifié. Une attaque de ce type est rendue possible à plusieurs niveaux⁶ des communications, dès qu'aucun mécanisme n'est mis en place pour authentifier les entités communiquant (on parle d'*ARP*, *IP*, *MAC*, ... *spoofing*). L'usurpation d'identité permet des attaques plus concrètes telles que les *MITM attacks* décrites ci-dessous.

Selective Forwarding : un nœud infecté ou malicieux introduit dans le réseau peut influencer celui-ci quand il est question de retransmettre des paquets (il effectue un travail de routage). Selon une certaine procédure créée pour manipuler artificiellement le trafic dans un sens profitant à l'attaquant, le nœud ne retransmet que les paquets qu'il sélectionne. Afin que ce ne soit pas trop évident de déceler le subterfuge, il n'arrête pas toutes les retransmissions.

Attaque de Sybil : un nœud malicieux gère plusieurs identités simultanément, propres ou en se faisant passer pour un autre nœud. Du point de vue des nœuds légitimes, ce sont donc plusieurs équipements distincts (simultanément ou représenté tour par tour). Une panne ou une attaque impactant la disponibilité d'autres nœuds peut ainsi être masquée par ce procédé.

5. https://www.cl.cam.ac.uk/~sps32/semi-inv_def.html

6. https://en.wikipedia.org/wiki/Spoofing_attack

Sinkhole/Blackhole attack : l'attaque consiste à consommer les ressources de nœuds voisins dans le rayon d'émission d'un nœud infecté. Cela se fait essentiellement en occupant le canal radio, brouillant les transmissions légitimes. Il en résulte une consommation d'énergie anormalement élevée pour les noeuds dans cette zone, attribuée aux retransmissions alors nécessaires.

Wormhole : cette attaque cible les protocoles de routages, introduisant une confusion dans le calcul des routes. Un nœud infecté va transmettre un paquet qu'il est censé router, directement vers un autre nœud éloigné en établissant un tunnel invisible pour le protocole de routage. Ainsi, des nouvelles routes vont apparaître, donnant l'illusion que les deux noeuds sont proches, ce qui pour un protocole de routage se basant sur la distance entre deux noeuds constituera une nouvelle meilleure route (qui ne l'est en réalité pas du tout).

Man-in-the-Middle attack (MITM) : forme d'eavesdropping où l'attaquant s'interpose entre deux noeuds voulant communiquer légitimement. Il opère de façon à ce que la communication ait lieu normalement du point de vue des deux autres intervenants, mais de sorte que chacun communique avec lui qui joue le rôle d'intermédiaire. Il peut alors accéder au contenu de tous les échanges et le manipuler pour en tirer des informations confidentielles et influencer le déroulement de la communication.

Hello-flood attack : inondation du canal par des messages inutiles d'annonces de voisin qui demandent une réponse de la part des nœuds qui le reçoivent (générant ainsi davantage de trafic).

3.5.3 Sur la couche support et application

Altération/exploitation des données : comme ces couches manipulent des données issues de ce que les équipements ont mesuré et capté, elles peuvent être sensibles et doivent être protégées. Un attaquant qui met la main sur ces données pourrait d'une part les corrompre et d'autre part les analyser et les exploiter afin de mener une autre attaque contre les utilisateurs. Par exemple, il pourrait dégager à partir des données enregistrées quelles sont les plages horaires pendant lesquelles un domicile n'est pas occupé et donc trouver une fenêtre de liberté pour une approche physique. De plus, ces données ne sont plus directement sous le contrôle de l'utilisateur (stockées dans les machines d'une entreprise), cela peut donc être problématique et mener à une atteinte à sa vie privée (revente illégale d'informations à une compagnie tierce, etc.).

Attaques de déni de service (*DoS, DDoS*) : les compagnies qui hébergent les services repris dans les couches support et application (traitement des données, accès aux serveurs, etc.) sont elles aussi susceptibles de se faire attaquer directement. Les attaques DoS (ou DDoS comme illustré avec Mirai en introduction de cette section) sont courantes et peuvent complètement interrompre la disponibilité de ces services par différent biais (congestion, mise à l'arrêt des machines).

Injection de code : l'attaquant utilise des failles dans les applications pour injecter et faire exécuter son propre code écrit à des fins malicieuses. Cela se fait notamment sur les bases de données (injection SQL). Une injection peut résulter en l'installation d'un malware, des requêtes illégitimes effectuées sur les bases de données, un gain de priviléges dans le système pour l'attaquant ou encore rendre possible du *cross-site scripting*⁷.

Sniffers/loggers : les attaquants récoltent un maximum des données utilisées et transitant (par exemple lors des échanges avec l'utilisateur par le biais de l'application) afin d'en tirer des informations de valeur (bien que cela soit rendu plus difficile par le chiffrement). Il peut s'agir d'identifiants, de mots de passes, d'emails, etc.

Session Hijacking : les attaquants exploitent des failles dans les processus d'authentification et gestion de session afin d'interférer dans les échanges et obtenir des accès privilégiés à des ressources. Cela se fait principalement via les cookies.

Ingénierie sociale : l'attaquant extrait des informations ou interagit directement avec l'utilisateur inaverti pour obtenir des identifiants ou autre en fonction de l'application à laquelle l'utilisateur accorde sa confiance.

7. https://en.wikipedia.org/wiki/Cross-site_scripting

3.6 Ressources, solutions et outils existants

3.6.1 Le travail d'OWASP

La fondation [OWASP](#) (*The Open Web Application Security Project*) est une organisation à l'échelle mondiale qui aspire à améliorer la sécurité informatique des systèmes et logiciels. L'IoT étant en plein essor, [un projet](#) a vu le jour sur la plateforme web, décrit comme « conçu pour aider les fabricants, développeurs et utilisateurs à mieux comprendre les problèmes sécuritaires liés à l'IoT, et pour permettre aux utilisateurs de prendre de meilleures décisions relatives à la sécurité quand ils déploient et travaillent avec des technologies de l'IoT ». Ainsi, plusieurs sous-projets plus spécifiques sont destinés à peupler le projet principal, certains étant encore actuellement en développement.

L'équipe travaillant sur le projet a établi [une liste](#) des principes relatif à la sécurité dans l'IoT. Plusieurs guides ont également été écrits, orientés chacun selon un point de vue différent : pour le fabricant, le développeur et l'utilisateur. D'autres ressources écrites sont disponibles, mais encore aucun outil développé autour de la sécurité n'a vu le jour comme c'est le cas pour d'autres domaines (les applications web particulièrement puisque originellement l'organisation ciblait ce secteur).

3.6.2 La standardisation de protocoles adaptés sécurisés

La sous-section 1.3 présentait les protocoles utilisables dans l'IoT, adaptés aux restrictions des équipements qui ne permettent pas toujours une compatibilité avec la *pile TCP/IP* classique. Lors de la conception de ces protocoles, la composante sécurité a été prise en compte et des mécanismes y ont été introduits pour l'assurer (ou alors des chercheurs ont fait ce travail par la suite).

6LowPAN propose déjà des mécanismes utiles pour garantir la sécurité [26]. Pour éviter les attaques par fragmentation, les options d'en-têtes *Timestamp* et *Nonce* sont un premier rempart. Des chercheurs ont également élaboré une version allégée de la suite de protocoles IPsec afin de garantir des communications sécurisées d'un nœud à l'autre dans un réseau 6LowPAN [20]. D'autres chercheurs présentés dans le même article ont proposé un système d'établissement de connexion sécurisée à partir d'un échange de clés.

RPL, utilisé pour le routage dans les réseaux de capteurs notamment au dessus de 6LowPAN, dispose également de mécanismes de protection. Il s'agit dans ce cas de garantir que quand un nœud sélectionne son parent pour rejoindre la topologie, ce dernier soit un nœud légitime (éviter le *spoofing*). Des solutions d'autres chercheurs sont présentées par Huichen [20] afin de sécuriser la génération des tables de routage dans RPL. Un exemple est un schéma de topologie authentifiée appelée TRAIL (*Trust Anchor Interconnection Loop*) qui permet la découverte et l'isolation des nœuds falsifiés et où chaque nœud peut, en envoyant un message particulier, valider son chemin jusqu'à la racine.

CoAP, tout comme son homologue HTTP, peut s'appuyer sur une couche intermédiaire plus basse pour assurer un transport sécurisé (chiffré pour préserver la confidentialité). Il s'agit de DTLS (*Datagram Transport Layer Security*), un équivalent de TLS plus adapté aux communications d'équipements restreint. La principale différence réside dans le fait que TLS soit basé sur TCP, au contraire de DTLS qui l'est sur UDP, toujours dans l'optique d'utiliser le minimum de ressources. CoAP offre le choix entre quatre modes relatifs à la sécurité : *NoSec*, *PreSharedKey*, *RawPublicKey* et *Certificate*.

3.6.3 La détection des équipements en réseau et de leurs vulnérabilités

Face à un réseau réel, disons local à une habitation possédant des équipements connectés (c-à-d orientée smart-home), il est difficile de se faire une idée de tous les objets y étant connectés et de quelles sont leurs fonctionnalités. Des outils sont nécessaires afin de découvrir le réseau et établir quels équipements peuvent potentiellement être la cible d'attaques. Il en existe une flopée ([sectools.org](#) en maintenant un classement fourni), libres ou commerciaux, possédant une interface graphique ou non, fonctionnels sur des OS variés. Outre ces logiciels complets aux champs d'action assez étendu, des outils de scanning plus spécifiques à des cibles (routeurs, serveurs, ...) et services (Web, SQL, mail, ...) sont trouvables dans des dépôts dédiés sous la forme de scripts ([Scanners-box](#)⁸ est dédié à leur référencement). Des exemples sont donnés ci-dessous, classés en deux catégories [19] en fonction de la manière dont le scan est effectué. Une attention particulière est portée aux outils dédiés à la détection des failles sur les équipements IoT d'un réseau s'il en existe qui soient dédiés.

8. <https://github.com/We5ter/Scanners-Box>

3.6.3.1 Le scan passif et la prise d'empreintes

L'objectif est d'en apprendre un maximum sur les équipements connectés au réseau sans les interroger directement. Cela se fait en écoutant toutes les transmissions passant sur un média utilisé par ces équipements, les capturant, les stockant et les agrégeant. Cela est donc assimilable à la pratique du sniffing. L'utilisation qui est faite de ces données varie en fonction de l'application : soit elles sont étudiées telles quelles car c'est le trafic qui est intéressant (**analyseurs de trafic**), soit elles sont utilisées pour faire de la « prise d'empreinte » (*TCP/IP stack, OS fingerprinting*) afin de caractériser les nœuds du réseau, soit les deux. Il peut également y avoir, suite à la phase d'écoute passive, des actions entreprises (par exemple si le processus a déterminé qu'un hôte présente une faiblesse exploitable).

Un analyseur de paquet populaire est Wireshark, qui utilise l'API *pcap* pour capturer les paquets transitant par le réseau sur lequel une interface désignée est placée en mode promiscuité (c-à-d fait remonter au système tous les paquets qui lui parviennent). Beaucoup d'autres utilitaires de capture (Kismet, tcpdump, ngrep, etc.) utilisent l'API *pcap*, implémentée dans la librairie *libcap* (systèmes UNIX) ou WinPcap (port de libcap vers Windows), il s'agit donc d'une ressource importante. Wireshark permet un filtrage assez fin des lots de paquets capturés (plus d'un millier de protocoles reconnus), permettant une grande flexibilité en fonction du but de la capture (debug, recherche d'une information, surveillance du trafic, etc.).

Les logiciels destinés à la capture d'empreintes et uniquement à cela sont assez peu nombreux (la plupart intègrent d'autres fonctionnalités actives). **p0f**⁹ est l'outil minimaliste le plus efficace. Il utilise diverses métriques extraites des en-têtes IPv4/6, des en-têtes TCP, des dynamiques observables dans le 3-way handshake de TCP et des données exploitables au niveau de la couche application (HTTP, SMTP, FTP, etc.). Combinées à une base de données de signatures récupérées expérimentalement, ces mesures permettent de déduire quels systèmes opèrent derrière les nœuds du réseau communiquant (là où un outil comme Nmap peut en être incapable). Outre cela, p0f est capable de déduire d'autres informations :

- présence d'un pare-feu, d'un système de NAT ou *load balancer*, de proxy applicatifs
- mesures du temps d'activité d'un système, sa distance (grâce au TTL), ses préférences au niveau de la langue, etc.
- les services utilisés par les clients et serveurs qui déclarent des champs comme X-Mailer ou User-Agent

D'autres outils existent, leurs fonctionnalités s'étendant au-delà de ce qui est considérable comme outil passif. On peut citer notamment :

- **Ettercap**¹⁰ utilise la capture de paquets pour intercepter des informations sensibles, et d'autres opérations avec pour but final d'effectuer une attaque de type *Man-In-The-Middle* (MITM)
- **Scapy**¹¹ et **Kamene** (fork du projet), des utilitaires en Python, permettent la capture et le reforgeage de paquets, la prise d'empreintes, le lancement d'attaques, etc. (le tout de façon modulable)

9. <http://lcamtuf.coredump.cx/p0f3>

10. <http://www.ettercap-project.org/>

11. <https://scapy.net/>

3.6.3.2 Le scan actif

Ces logiciels tentent d'établir des connexions avec les hôtes à analyser, à la recherche d'un service répondant donc potentiellement ouvert, non protégé et dont il est possible de soutirer des informations. Une liste de numéros de ports pertinents est construite par le scanneur, utilisée par ce dernier lors de l'analyse des hôtes qu'il trouve dans le réseau. Cette technique basique est appelée **balayage de port**. Se servant de ces scanneurs comme appoint, les **scanneurs de vulnérabilités** vont plus loin en tentant de découvrir pour chaque hôte analysé les failles de sécurité auxquelles ils sont exposés (au niveau d'une application, du système d'exploitation ou des interfaces réseaux).

L'outil **Nmap**¹² est dédié au balayage de ports et à l'acquisition directe d'informations (version, composants matériels utilisés, etc.) concernant le système et les services ouverts d'un hôte analysé. La détection de vulnérabilités est rendue possible grâce aux interactions scriptées avec la cible en utilisant le Nmap Scripting Engine (NSE), un exemple de module Nmap y étant dédié est **Vulscan.nse**. Nmap a d'autres fonctionnalités pratiques telles qu'une interface graphique complète (**Zenmap**) et la personnalisation des formats de sorties pour les résultats (filtrage, langages balisés, etc.).

OpenVAS¹³ est une branche du projet **Nessus**, ce dernier étant passé sous licence commerciale. Ce sont les frameworks pour la détection de vulnérabilités sur un réseau les plus populaires. Tenable[®] (qui possède Nessus) propose des **services de scan dédiés aux équipements IoT** utilisés dans l'industrie, mais OpenVAS n'offre rien de similaire.

Le cas de **Lynis**¹⁴ est un peu particulier. Effectivement, en opposition aux scanneurs en réseau comme OpenVAS, il s'agit d'un scanneur de vulnérabilités à lancer sur l'hôte à analyser. Cela permet une inspection beaucoup plus en profondeur des services et des vulnérabilités qu'il présente. Il tourne sur presque tous les systèmes UNIX et même sur un système limité tel qu'une carte Raspberry Pi ou un équipement relatif à l'IoT qui est capable de supporter le même type de charge. Lynis peut donc être un outil intéressant si on possède le contrôle de l'équipement à analyser (légitimement ou non).

Dédié à la détection des vulnérabilités des équipements IoT, **IoTSeeker**¹⁵ est un outil libre et écrit en Perl, supporté par les systèmes Linux/Mac OS. Il est axé sur les vulnérabilités d'authentification, la découverte de mots de passe laissés par défaut par les utilisateurs imprudents (sur base des listes des identifiants par constructeur évoquées dans la sous-section 3.1).

RouterSploit¹⁶ est un framework dédié à la détection et l'exploitation de failles dans les systèmes embarqués. Comme son nom l'indique, la cible est davantage les routeurs que les équipements de l'IoT, cependant quelques modules ciblent notamment les caméras et d'autres équipements connectés insolites (projecteur, laveur de laboratoire). L'outil de pénétration **Nettacker** développé par OWASP, dont le but est le soutirage et l'agrégation d'informations d'une cible pour en dresser une liste des failles de sécurité, promet également la sortie future d'un module spécifique à l'IoT.

12. <https://nmap.org/>

13. <http://www.openvas.org/>

14. <https://cisofy.com/lynis/>

15. <https://github.com/rapid7/IoTSeeker>

16. <https://github.com/threat9/routerSploit>

4 Développement d'une application *IoTMonitor* - introduction et concepts

4.1 Présentation générale

4.1.1 Contexte et environnement ciblé

Ce travail englobe la thématique de la sécurité des équipements en réseau dans l'environnement domestique, en mettant l'accent sur les différents types de pare-feux. Effectivement, ces derniers peuvent palier en partie aux faiblesses récurrentes des équipements IoT qui peuplent les réseaux domestiques. L'aboutissement final du projet est de développer un outil qui puisse contribuer à la protection de ces réseaux. Cependant, l'analyse de trafic avancée (au delà du simple pare-feu à état) telle qu'utilisée dans les solutions commerciales est une technologie non triviale qui sort des limites de ce projet (théorie de *machine learning*, construction de base de données de signatures, etc.). À l'inverse, se limiter à un simple pare-feu à état ciblant un protocole de l'IoT peut sembler réducteur vis-à-vis de toutes les menaces planant sur ces équipements qu'un tel pare-feu ne pourrait contrer.

La sous-section 3.6 met en évidence quelques outils qui peuvent aider à analyser le réseau domestique et détecter les failles exploitables des équipements qui le peuplent. Ces *scanners*, conjugués à des systèmes de prévention d'intrusion *NIDS*, permettraient d'améliorer grandement la sécurité du réseau domestique, tout en n'effectuant aucune tâche de filtrage direct du trafic comme le ferait un pare-feu en bordure du réseau. En découle l'idée motrice de l'application à développer : prodiguer un outil permettant l'**analyse, la surveillance et l'agrégation des caractéristiques des équipements (adresses IP/MAC, ports ouverts, etc.)** d'un réseau domestique, basé sur des **scanneurs** existants ciblant **differents aspects** de sa sécurité. Le système doit donc être conçu de façon modulaire, de sorte que l'**intégration d'un nouveau scanneur** ne requiert comme travail que la traduction (parsing) de la sortie dudit programme vers des actions et éléments interprétables par l'application.

Par exemple, l'application pourrait intégrer des scanneurs tels que présentés dans la sous-section 3.6. Parmi ceux-ci *nmap*, *IoTSeeker*, etc. qui interrogent directement les équipements (dits actifs) et des programmes de prise d'empreintes/analyse de trafic tels que *Snort*, *p0f*, etc. (dits passifs). Chacun de ces programmes fournit un feedback sur sa sortie, que ce soit une fois son exécution terminée ou de façon continue jusqu'à interruption manuelle de l'exécution. Une fois parsées, les informations obtenues peuvent servir à alimenter une représentation virtuelle du réseau maintenue par l'application (renseignant les caractéristiques des équipements qui le peuplent) ou constituer une alerte à faire remonter jusqu'à l'utilisateur si une faille ou menace est détectée.

Par programme il est entendu : *un programme ou script installé et fonctionnel sur le système, exécutable depuis un terminal et dont la sortie est récupérable*. Le framework d'intégration définit comment ajouter ces programmes à l'environnement de l'application, prodiguant une surcouche telle qu'il n'est nécessaire de définir que les éléments suivants :

- la **commande** liée au programme telle qu'utilisée pour y faire appel depuis un terminal
- quels **paramètres du programme** utilisés parmi ceux passables en ligne de commande et la façon d'y associer des valeurs (*schéma* de paramètres)
- la **méthode de traitement** de la sortie du programme (parsing) pour en inclure les résultats dans l'application

4.1.2 Objectifs de l'outil

4.1.2.1 La surveillance automatique du réseau domestique

Une manipulation aisée des programmes à utiliser pour surveiller le réseau est essentielle. L'utilisateur doit pouvoir sélectionner et configurer les programmes qu'il souhaite pour défendre son réseau parmi ceux disponibles. Une fois cette sélection opérée, les programmes choisis devraient être maintenus dans une structure où leur exécution, la récupération de leur sortie et le parsing de ces dernières sont entièrement automatisés. L'utilisateur n'ayant alors qu'à agir sur cette structure pour contrôler l'état de la surveillance de son réseau. La gestion du travail des programmes qui y sont maintenus est entièrement automatisée et ne requiert pas d'interaction avec l'utilisateur. L'application peut alors agir de façon autonome et effectuer sa tâche de surveillance en fond continuellement. Ladite structure est un objet appelé **Routine** dans l'application.

4.1.2.2 La représentation du réseau

Puisque n'importe quel programme peut potentiellement être intégré dans l'environnement de l'application, les sorties sont totalement arbitraires. Cependant, il est très probable que celles-ci contiennent des informations relatives aux hôtes sur le réseau analysé : adresses MAC, IP, nom d'hôte, etc. Un objectif de l'application est de regrouper toutes ces informations afin de construire une carte virtuelle du réseau la plus fidèle possible. Cette carte est appelée **Netmap** et doit être consultable par l'utilisateur de façon à ce qu'il ait une vision synthétique mais complète de l'état de son réseau. L'utilisateur doit également pouvoir interagir avec cette carte en créant et modifiant des informations associées aux hôtes, afin d'assister ou corriger la récolte automatique d'informations.

4.1.2.3 La détection de menaces

Certains programmes intégrés à l'application sont conçus pour trouver des failles existantes aux hôtes du réseau. Une fois les sorties de ces programmes parsées, les menaces détectées doivent être remontées jusqu'à l'utilisateur et archivées dans l'application. Les notifications à l'utilisateur, s'il n'est pas directement en train d'utiliser l'application, se font par un email récapitulatif de la menace trouvée. Pour en consulter les détails par la suite, les menaces associées à un hôte doivent être consultables dans l'application même.

4.1.2.4 La conception d'un framework d'intégration

Puisque l'application fonctionne en se basant sur des programmes extérieurs arbitraires, il est nécessaire de définir un cadre assez rigoureux pour les intégrer au système de sorte à pouvoir manipuler leur exécution et leur passer des paramètres. Cela se fait via une surcouche d'abstraction désignée par le terme **Module**. Dans les faits, il s'agit d'une classe à écrire en Python héritant d'une classe abstraite (faisant office de framework). Deux types de Modules sont envisagés pour abstraire un programme en fonction de son schéma d'exécution, décidant de la classe abstraite à sous-classer. Ces deux archéotypes sont détaillés dans la sous-section 4.3.

4.1.2.5 L'utilisation en environnement restreint

L'application doit pouvoir être utilisée telle quelle depuis un équipement connecté au réseau domestique, similaire ou supérieur en terme de ressources à une carte Raspberry PI. Également, l'application devrait pouvoir être pleinement utilisée dans un environnement non graphique, par exemple sur la carte Raspberry depuis une connexion SSH distante. Effectivement, comme le travail de surveillance effectué par l'application nécessite une machine continuellement active, il est préférable pour l'utilisateur qu'elle soit peu consommatrice. Une carte Raspberry PI 3 dédiée à l'utilisation de l'application devrait être suffisante.

4.2 Composants du système de surveillance

Afin de mieux visualiser les différents éléments qui composent le cœur de l'application, une maquette d'interface graphique a été élaborée. Celle-ci représente ce à quoi devrait correspondre l'écran de gestion principal de l'application. Effectivement, afin de pouvoir être utilisée dans un contexte restreint (tel que sur une carte Raspberry PI), l'application a d'abord été dotée d'une interface en ligne de commande en priorité (voir point 5.2.6). Une partie des composants principaux sont repris dans la figure 9.

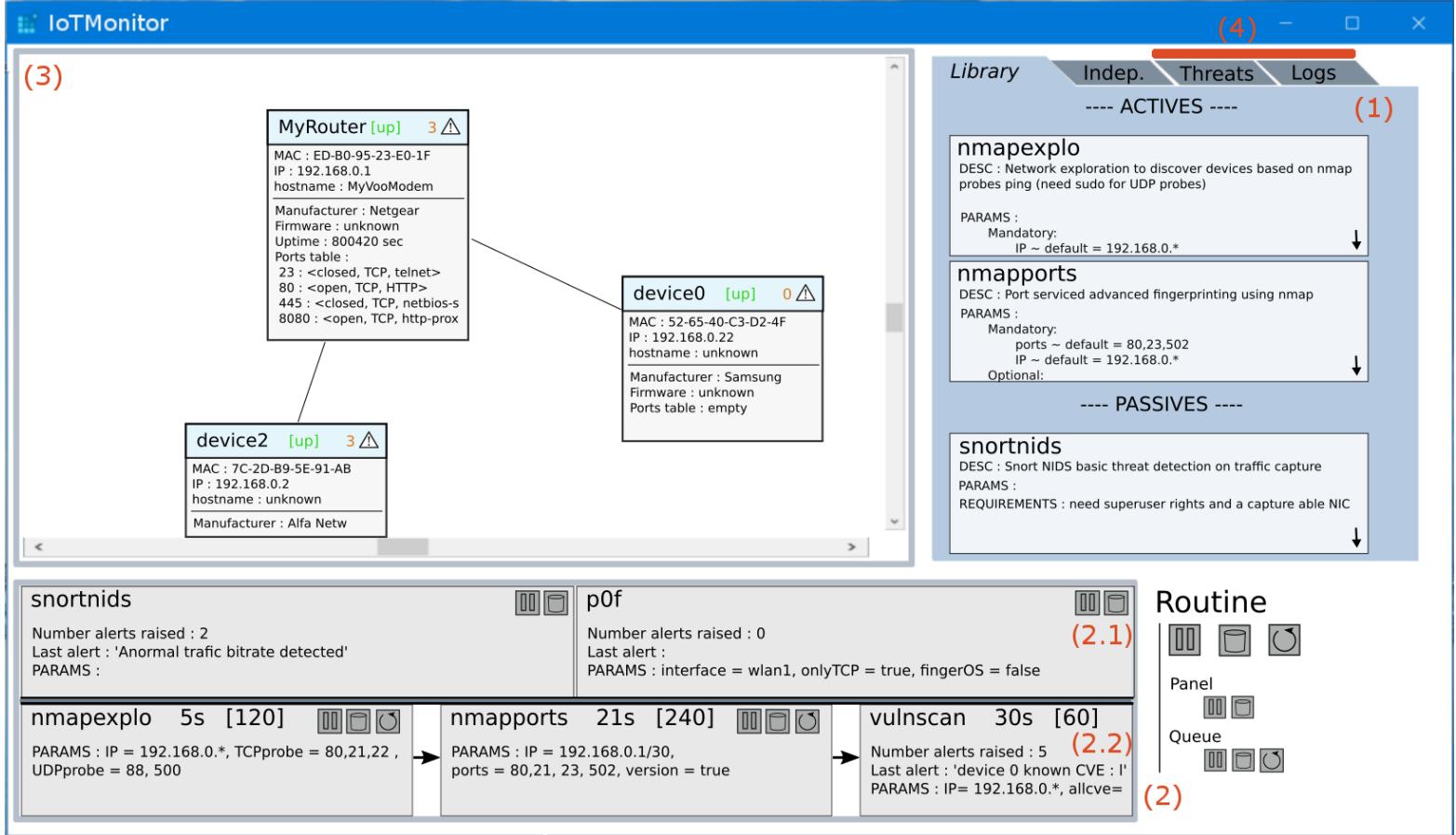


FIGURE 9 – Maquette GUI de l'application en reprenant les principaux éléments

4.2.1 Modules actifs et passifs - abstraction des programmes

Les Modules constituent la couche d'abstraction pour les programmes sous-jacents et sont régis par un framework spécifique à l'archéotype attribuable au programme. On distingue deux archéotypes de programmes selon leur schéma d'exécution propre :

- Les programmes dits *actifs* par analogie avec les scanneurs actifs : une fois appelés, ils effectuent un travail qui se termine après un temps fini. Les résultats de ce travail sont alors envoyés dans le flux de sortie standard ou vers un fichier. Cette information est par la suite interprétable humainement ou parsée automatiquement.
- Les programmes dits *passifs* par analogie avec les scanneurs passifs, les programmes de prise d'empreintes et d'analyse du trafic (NIDS). Une fois appelés, ils s'exécutent en continu pour traiter des informations récupérées au cours du temps (du trafic notamment). N'interrogeant pas directement les équipements, ces programmes attendent que ceux-ci émettent eux-mêmes de l'information à traiter. Les résultats sont également transmis vers un flux de sortie, mais étant donné que l'exécution ne s'arrête pas cela est fait en continu de façon irrégulière et non pas à la terminaison de l'exécution comme pour un programme actif.

De nouveaux programmes peuvent ainsi être intégrés et utilisables dans l'application, à condition d'écrire une classe héritant de la classe abstraite correspondant soit aux modules actifs, soit aux modules passifs en fonction de la nature du programme cible. Dans ces classes parents, la façon dont l'exécution du programme sous-jacent doit être traitée est déjà encadrée en implémentée sur base de l'assumption faite de l'archéotype. L'implémentation de la gestion de l'exécution du programme sur base du schéma est la suivante :

- **Module actif** : un thread est lancé pour dissocier le fil d'exécution général de l'application. Dans ce thread, un processus fils exécutant le programme est instancié, avec sa sortie récupérable. Une fois l'exécution terminée, le thread se termine en appelant une fonction de callback sur la sortie, la définition de cette fonction étant spécifique au parsing de la sortie programme.
- **Module passif** : le travail est divisé en deux parties. La première se charge de lancer un processus fils exécutant le programme dont la sortie est redirigée vers un pipe tandis que la seconde est chargée de lire le contenu de ce pipe et d'appeler la fonction de parsing dessus à intervalle régulier.

Les Modules sont manipulables comme des unités dans l'application. Ils sont référencés dans la Librairie qui est le point d'entrée à partir duquel les instancier. Elle est représentée dans la maquette par le label (1). À partir de là, une instance de Module peut soit être insérée dans la Routine de surveillance si on souhaite son utilisation régulière automatisée, ou dans un ensemble à part dit indépendant, où il sera exécuté directement une unique fois.

Plusieurs modules distincts peuvent abstraire le même programme, mais dont les utilisations spécifiques sont différentes. Par exemple, le programme *nmap* est accompagné du NSE (Nmap Script Engine), un framework permettant de coupler des scripts variés à l'exécution d'un scan nmap. Nmap peut donc être utilisé en combinaison avec différents scripts dont les objectifs, paramètres et sorties sont très hétérogènes. Cela implique que des couches d'abstraction distinctes devraient être écrites afin de les représenter par différents Modules dans l'application.

En terme d'implémentation, les éléments imposés par le framework que chaque implémentation de Module doit fournir sont :

- Un id alphanumérique unique parmi tous les autres Modules déjà référencés dans la Librairie
- Une description textuelle du rôle du Module
- L'archéotype attribuable au Module : à considérer comme actif ou passif
- La commande qui permet d'appeler l'exécutable sur le système (vraisemblablement dans le PATH)
- Le schéma de paramètres : une définition des paramètres passables à la commande et la façon de l'y injecter (et leur ordre relatif), chacun identifié par un code alphanumérique et présentant les attributs suivants :
 - un indicateur booléen indiquant s'il est indispensable (toujours injecté dans la commande)
 - une valeur par défaut, qui sera utilisée si le paramètre est indispensable mais non fourni
 - le flag associé à l'argument du programme en ligne de commande si existant
- Additionnellement, une description textuelle de chaque paramètre
- La méthode de parsing, fonction traduisant la sortie textuelle du programme vers des éléments de l'application

Le champ d'action d'un Module conforme au framework est le suivant :

1. Construire la commande d'appel au programme en utilisant le schéma de paramètres et les valeurs courantes de ces paramètres pour l'instance du Module, telle qu'elle serait écrite dans un terminal
2. Dissocier le fil d'exécution en exécutant la commande construite dans un nouveau thread
3. Maintenir une référence vers ce processus de sorte à pouvoir communiquer avec (signal d'interruption, etc.)
4. Récupérer les informations que le programme renvoie sur sa sortie standard et ses erreurs
5. Parser cette sortie et la traduire vers l'application afin d'y intégrer l'information :
 1. **Logging** : informations et événements à inclure dans les logs de l'application (debug, ...)
 2. **Alertes** : menace détectée par le programme, à ne pas simplement logger (associer à un équipement particulier, création et envoi d'un mail au propriétaire, ...)
 3. **Modifications** : un Module a l'accès en lecture et écriture sur la carte du réseau et ses éléments, permettant leur mise à jour automatique avec les nouvelles informations obtenues

Pour plus de détails concernant l'implémentation concrète du système de Modules, se référer à la sous-section 4.3.

4.2.2 Routine - automatisation de la surveillance

La Routine est essentiellement un conteneur pour les instances de Modules que l'utilisateur sélectionne pour surveiller son réseau. En agissant sur la Routine, tous les Modules qu'elle contient peuvent être manipulés d'un coup. Elle permet notamment d'en contrôler l'exécution, automatisant ainsi le lancement de tous les Modules sélectionnés par l'utilisateur. Un état **running** est associé à la Routine, reflétant l'état d'exécution des programmes sous-jacents aux Modules qui la composent. À chaque instance de Module dans la Routine, un id alphanumérique unique appelé **setid** est associé.

La Routine est indiquée dans la maquette par le label (2). Elle maintient deux sous-ensembles vers lesquels les modules sont aiguillés en fonction de leur archétype : le **Panel** des passifs (2.1) et la **Queue** des actifs (2.2). Cette division est directement liée au fait qu'il est nécessaire de traiter différemment l'exécution de Modules qui présentent des schémas différents : un Module actif devra être exécuté à répétition afin d'avoir un impact dans la surveillance du réseau. À l'inverse, un Module passif n'a pas besoin d'être relancé pour effectuer son travail jusqu'à son interruption depuis l'extérieur. Les deux ensembles gèrent donc de manière différente l'exécution des Modules contenus, mais maintiennent tous deux un état binaire **running**. L'état **running** de la Routine est mis à vrai dès lors que l'un des deux sous ensemble est en activité.

Le Panel maintient les modules respectant l'archétype *passif*. Son état commutable **running** correspond à l'exécution des instances de Module qu'il contient. Quand **running** est passé à vrai, chaque Module du Panel déclenche l'exécution de son programme sous-jacent. Quand le Panel est repassé en pause, tous ces processus vivants sont terminés (signal d'interruption).

Le but de la Queue est d'organiser la séquence des modules qui seront lancés. Pour cela, en plus du **qid** attribué à chacun de ses modules, un timer d'expiration est initialisé (à une valeur fournie par le module et modifiable). Il est décrémenté chaque seconde jusqu'à valoir 0, alors le module est lancé (entendu l'exécution du programme sous-jacent), et son timer est réinitialisé. L'ensemble des modules de la Queue peut alors être visualisé comme une file, où les modules sont agencés selon leur temps restant. On visualise ainsi la séquence d'exécution prochaine, totalement automatisée.

Comme énoncé lors de son introduction, la Routine est composée de deux ensembles de modules : Panel et Queue. Le premier est destiné à accueillir des modules respectant l'archétype **passif**, tandis que le second s'attend à des modules présentant l'archétype **actif**. Sur base de cette hypothèse, un contrôle cohérent de leur exécution peut être établi : par exemple l'action de *mettre en pause* un module passif dans le Panel et actif dans la Queue a une sémantique différente. Mettre en pause un module passif équivaut à envoyer un signal d'interruption au processus sous-jacent tournant en continu en arrière-plan. Mettre en pause un module actif dans la Queue équivaut à ne plus décrémenter son timer d'expiration associé, au terme duquel il était censé lancer une exécution du programme qu'il abstrait.

L'application référence tous les modules disponibles dans une **Librairie** de modules. Il ne s'agit, en arrière-plan, que d'un simple fichier XML décrivant tous les modules qui doivent être chargés dans un format strict. Afin d'intégrer de façon constante un module pour qu'il soit utilisable dans l'application, deux options sont possibles : soit écrire manuellement la description du module et l'ajouter au fichier correspondant (plusieurs librairies de modules peuvent être envisagées), soit simplement appeler une méthode dédiée sur une instance du module qui en créera le descriptif automatiquement. Si le module a été correctement écrit en suivant le strict minimum imposé par le framework, le descriptif du module sera ajouté à la librairie courante et le module facilement manipulable dans l'application (pour par exemple l'ajouter à la Routine de surveillance).

4.2.3 Netmap - recensement des équipements du réseau

La carte du réseau est le résultat de l'agrégation des informations parsées des sorties de programmes de la Routine (donc obtenue automatiquement) et des entrées manuelles de l'utilisateur. Il s'agit de résumer au mieux d'une part les informations récoltées concernant les équipements (adresses MAC, IP, hostname, ports ouverts, etc.) et d'autre part les alertes découvertes associées à un équipement particulier. Une alerte serait par exemple une faille connue pour la version d'un service qu'on découvre comme utilisée sur un port de l'équipement. Cette carte virtuelle du réseau a pour objectif de fournir une vue d'ensemble du réseau domestique limitée au LAN pour l'utilisateur, afin qu'il puisse y exercer un contrôle plus pertinent. Elle a pour objectif de représenter au mieux l'état actuel du réseaux et des hôtes qui le peuplent.

La carte du réseau est l'élément vers lequel doivent converger toutes les informations concernant le réseau, obtenues grâce aux programmes sous-jacents ou entrées par l'utilisateur lui-même. Elle a pour objectif de les agréger pour produire une représentation interne du réseau domestique cohérente, comme représenté dans la maquette par (3). Les équipements physiques y sont représentés par des **Instances Virtuelles** (IV), chacune destinée à correspondre à un unique équipement et regroupant un maximum des informations le concernant. Cet ensemble d'IVs est maintenu dans la **Netmap**, où elles sont indexées par un identifiant `mapid` attribué automatiquement `device0`, `device1`, ... modifiable par l'utilisateur s'il le désire en fonction de ce qu'est réellement l'équipement pour lui.

Les informations stockées dans les IVs sont appelées des **champs**. Ceux-ci peuvent être créés et remplis de façon automatisée par les modules (c'est un travail de la fonction de parsing des sorties à définir pour chaque module), mais également par l'utilisateur lui-même s'il souhaite étayer les informations concernant ses équipements. Quelques champs sont plus importants que d'autres, car plus pertinents dans un processus d'identification de l'équipement concerné par un paquet d'informations. Par exemple, un scanner de ports des équipements du réseau envoie sur sa sortie l'IP de chaque équipement analysé et une table des ports ouverts. Pour associer cette information à chaque IV concernée, il va falloir identifier dans la Netmap l'IV unique qui renseigne déjà cette IP. S'il n'y en a pas, une nouvelle IV est créée pour contenir l'information. Il est possible que l'équipement réel était en effet déjà référencé dans la Netmap, mais son champ IP vide. Il s'agit donc d'un problème non trivial, une première étape pour le résoudre est de définir dans quel ordre certains champs sont pertinents pour évaluer une correspondance.

Les champs principaux sont les suivants (par ordre de pertinence) :

1. Adresse MAC (unique pour chaque équipement, sauf en cas de spoofing)
2. Adresse IP (allouée généralement dynamiquement)
3. Nom d'hôte `hostname` (aucune garantie d'unicité, mais assez probable)

D'autres champs, considérés comme divers et auxquels de nouveaux peuvent venir se greffer à volonté :

- Fabriquant
- Modèle
- Firmware
- ...

Afin de limiter ce besoin d'identifier l'instance virtuelle relative à une information obtenue du réseau, une mécanique simple est prévue. Quand un module est lancé, il est possible de lui fournir en paramètre quelles sont les IVs auxquelles l'exécution est relative. Cela permet deux améliorations : accéder à des informations sur les équipements déjà connues pour paramétriser l'exécution du programme et à interpréter sa sortie en fonction d'IVs déjà connues (donc de ne pas avoir à chercher une correspondance). Cet aspect est à gérer dans l'implémentation de chaque module.

À chaque instance virtuelle est également associée une Table des ports. Celle-ci maintient de l'information pour chaque port qui a été détecté. L'indexation se fait simplement par le numéro associé au port, et les valeurs principales à y associer sont le nom du service correspondant, le protocole de transport utilisé et son état (`open`, `closed`, ...). Il y a également la possibilité d'y associer des informations diverses à volonté telles que la version du service, le taux de confiance en cette information, etc.

Une entrée dans cette table est donc de la forme :

`numPort : <service, protocole, état>, divers {(serv_version, val), (conf, val), ...}`

4.2.4 Centre d'événements - logging et lancement d'alertes

On distingue deux catégories d'informations qui doivent être enregistrées par l'application : les logs purs et les événements représentés par un objet **Event**. Les logs sont de simples informations à écrire dans un fichier ou une sortie donnée, tels que du debug, des codes de retour des programmes et autres informations lors de leur exécution, etc. Les Events sont quant à eux capturés par l'application et instanciés pour être manipulés par la suite. Les logs interviennent principalement dans le cadre de l'exécution des programmes sous-jacents aux modules, et sont donc utilisables depuis l'endroit de leur définition. La place de ces éléments dans la maquette est indiquée par (4).

Les Events peuvent être de deux natures, et constituent un moyen pour les modules de communiquer les modifications qu'ils effectuent sur les éléments de l'application. Soit ce sont des événements de modification, soit de menace. Le premier cas fait référence aux modifications sur la Netmap et ses instances virtuelles, notamment la création d'une IV ou un ajout ou changement de valeur d'un de ses champs. L'objet Event contient principalement des informations telles que l'ancienne valeur et la nouvelle, quel module a fait la modification. Un Event de menace est utilisé pour décrire une détection de menace au niveau de la sécurité du réseau domestique. Elle maintient des informations sur la nature de la menace, le possible patch à appliquer, le niveau de danger, etc. qui doivent être remplies par le module qui lance l'alerte.

L'élément de centralisation des Events de l'application est appelé **EventCenter**. Il est accessible par un simple appel depuis n'importe où dans l'application et permet de créer des Events qui y seront enregistrés. Il est alors possible d'appliquer des fonctions de recherche et filtrage sur ces événements, par exemple pour trouver toutes les modifications sur une instance virtuelle donnée. Un nombre constant limite le maximum d'événements gardés en mémoire, les plus anciens laissant leur place aux plus récents.

Pour éviter la perte d'événements importants concernant des équipements du réseau de la sorte, il est également possible d'accéder à l'EventCenter en passant par la Netmap. Un Event créé par ce biais, s'il concerne une IV de la Netmap, sera conservé de façon permanente dans la Netmap en plus de la journalisation temporaire dans l'EventCenter. L'indexation des Events dans la Netmap se fait simplement par le même index d'IV (son **mapid**). L'utilisateur peut ainsi facilement visualiser les modifications et menaces impliquant directement ses équipements.

4.3 Modèle d'abstraction des logiciels

4.3.1 Modèle général d'abstraction

Bien que l'on considère deux archétypes différents, les modules actifs et passifs doivent implémenter plusieurs méthodes communes. On définit donc ce qui équivaut à une interface générale dont deux classes abstraites héritent. Le framework pour l'intégration des modules est défini de cette façon. La figure 10 reprend ce framework, illustrant quelles méthodes doivent être implémentées pour que l'application puisse utiliser le nouveau module écrit.

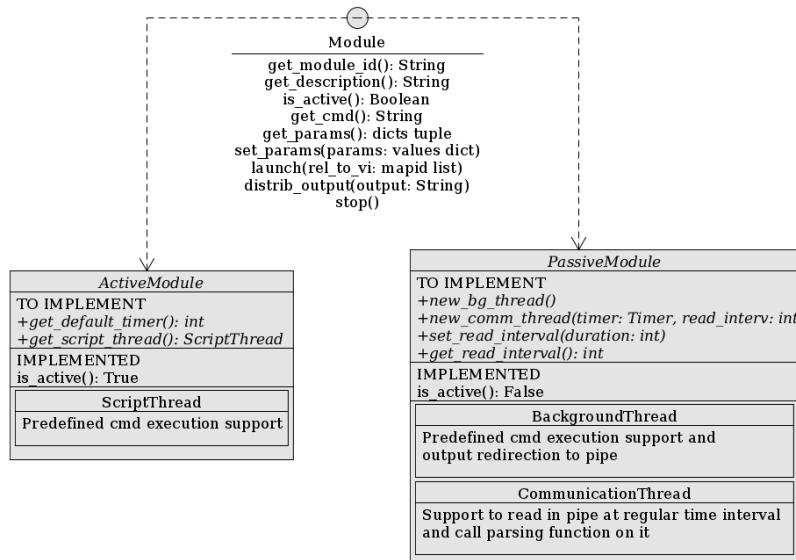


FIGURE 10 – Structure orientée objet du modèle d'abstraction

À noter que chaque module devrait garder une référence vers les threads dans lesquels il a lancé une exécution de son programme sous-jacent (plus de détails aux points suivants). Les méthodes remarquables de l'interface *Module* ont la sémantique suivante :

- *get_cmd()* : retourne l'alias qui sert à appeler le programme en CLI (comme lors d'une utilisation normale)
- *get_params()* : obtention de 3 éléments : paramètres actuels, par défaut et leur description textuelle
- *set_params()* : ajustement de quels paramètres et de leur valeur à fournir à une exécution du programme
- *launch()* : lance une exécution du programme dans un nouveau thread, avec les paramètres courants du module. Les définitions pour un schéma d'exécution classique sont déjà implémentées dans les classes abstraites de l'archéotype correspondant (ce sont des objets instanciables avec certains paramètres).
- *distrib_output()* : méthode de parsing appelée sur la sortie de l'exécution d'un programme, effectue le travail de traduction vers les éléments de l'application
- *stop()* : interrompt l'exécution de tous les programmes sous-jacents dans les threads liés à l'instance du module

4.3.2 Design pour un module actif

Le schéma d'exécution classique pour l'archéotype actif est simple : l'appel à *launch()* doit lancer l'exécution du programme sous-jacent avec les paramètres actuels du module et dans un nouveau thread, auquel une fonction de callback (*distrib_output()*) est passée en paramètre. À la fin de l'exécution, la sortie du programme est récupérée et l'appel à la fonction de callback est effectué sur cette sortie (donc le parsing se fait dans la fin du fil d'exécution du thread). Ce comportement peut être obtenu grâce au support déjà implémenté dans la classe abstraite *ActiveModule* dont un module concret doit hériter. Il reste cependant des méthodes spécifiques au module qui doivent y être implémentées (notamment celles de l'interface *Module*). La figure 11 résume cette information.

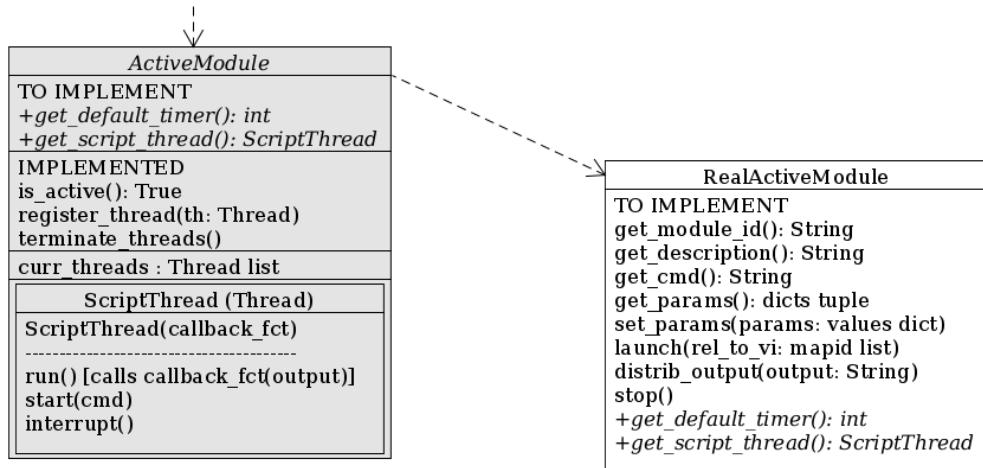


FIGURE 11 – Framework d'écriture d'un module actif

Le code d'un module actif devrait être structuré de la sorte (si utiliser les facilités déjà implémentées est désiré) :

1. *get_script_thread()* doit retourner une instance de *ScriptThread* paramétrisée avec la fonction *distrib_output()* du module.
2. *launch()* construit la commande d'appel au programme *cmd* depuis les paramètres actuels du module, telle qu'elle serait invoquée en ligne de commande (en utilisant *get_cmd()* + les paramètres actuels dans *get_params()*).
3. *launch()* instancie le conteneur de l'exécution avec *get_script_thread()* et invoque sa méthode *start()* avec *cmd*, ce qui va lancer un nouveau thread dans lequel le processus du programme sous-jacent est créé. À la fin de son travail (*run()* est bloquant dessus), la méthode *distrib_output()* est appelée sur sa sortie.
4. *launch()*, après avoir créé et dissocié le thread du fil d'exécution, l'enregistre dans l'ensemble des exécutions courantes du module avec *register_thread()*.

Additionnellement, la méthode *get_default_timer()* renseigne la valeur par défaut que devrait prendre le timer d'expiration d'une instance du module placée dans la Queue.

4.3.3 Design pour un module passif

Pour un programme correspondant à l'archétype passif, le schéma d'exécution est complexifié car il faut pouvoir récupérer les sorties du programme au fur et à mesure qu'elles sont produites et de façon régulière. Effectivement, on ne peut pas attendre la terminaison du programme qui marque le fait qu'il y a quelque chose à lire sur sa sortie puisqu'il n'est pas censé terminer son exécution de lui-même. La solution choisie est de séparer le schéma d'exécution en deux parties. L'une contrôle le processus sous-jacent et redirige sa sortie vers un pipe temporaire pendant que l'autre partie y lit les sorties à intervalle régulier et les parse. Comme pour l'archétype actif, des facilités sont fournies pour gérer ce travail, illustrées par la figure 12.

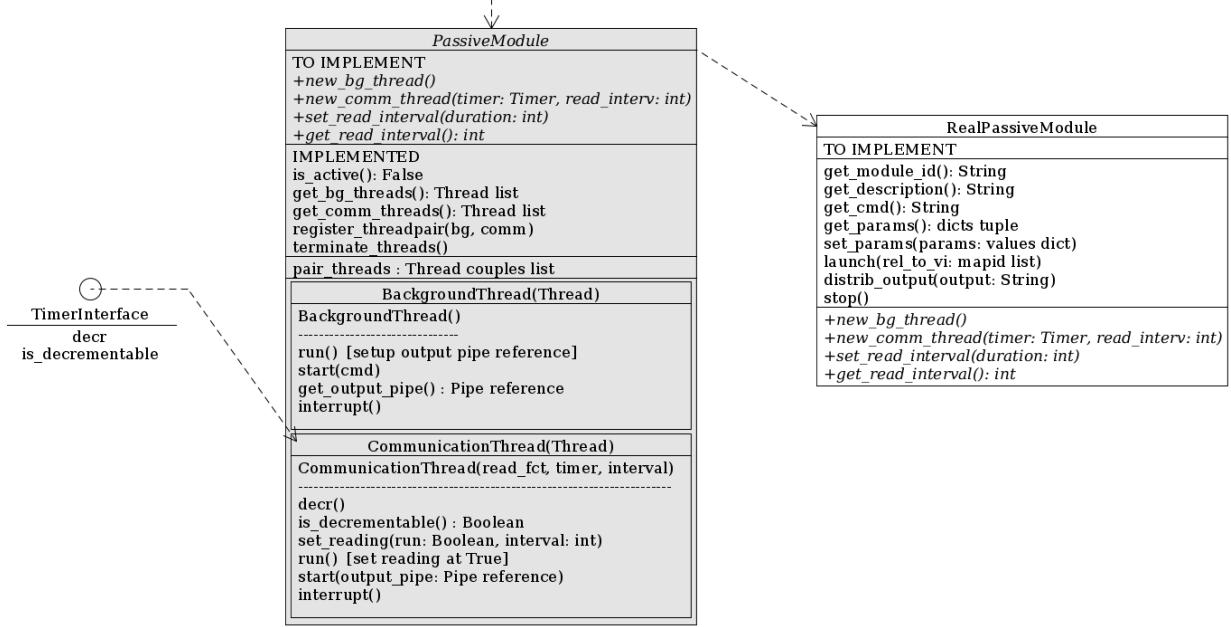


FIGURE 12 – Framework d'écriture d'un module passif

Pour utiliser correctement ces facilités, l'implémentation du module devrait respecter ces recommandations :

1. `new_bg_thread()` et `new_com_thread()` retournent respectivement une instance de `BackgroundThread` et `CommunicationThread`. Cette dernière doit être paramétrisée avec un objet `Timer`, l'intervalle de temps auquel la lecture doit avoir lieu et la fonction de parsing `distrib_output()` appelée sur la sortie lue
2. `set_read_interval()` et `get_read_interval()` sont relatifs à l'intervalle entre deux lectures qui doit être > 0
3. Comme pour les modules actifs, `launch()` devrait construire la commande `cmd` telle qu'utilisable en CLI
4. Dans un premier temps, `launch()` appelle la méthode `start()` d'une instance de `BackgroundThread` en lui passant `cmd` pour démarrer le processus du programme sous-jacent, tout en redirigeant sa sortie vers un pipe alors créé à cette fin (`output_pipe`). Une référence vers ce dernier est maintenue.
5. En second lieu, `launch()` devrait attendre que le processus ait terminé son initialisation, c'est-à-dire que `output_pipe` soit créé et accessible en lecture. Alors le `CommunicationThread` peut être instancié en le lui passant en paramètre.
6. Finalement, `launch()` devrait enregistrer cette paire d'objets Thread dans la liste prévue à cet effet avec un appel à `register_thread_pair()`

À noter qu'ici, la méthode de `BackgroundThread run()` lançant le sous-processus ne bloque pas jusqu'à terminaison (ce serait une perte de ressources). Également, la méthode `run()` de `CommunicationThread` fait passer l'état de l'objet à *décrémentable* puis termine le fil d'exécution du thread. C'est quand son timer tombe à 0 qu'un nouveau "dumb thread" est créé avec pour unique tâche de faire la lecture et le parsing. Cela se justifie par le fait que la méthode qui fait l'appel à la décrémentation (propre à l'objet `Timer` régie l'application) ne devrait pas effectuer ce travail. En effet, il pourrait perturber la gestion de l'écoulement du temps s'il est trop lourd et effectué dans le même fil d'exécution.

5 Implémentation de l'application

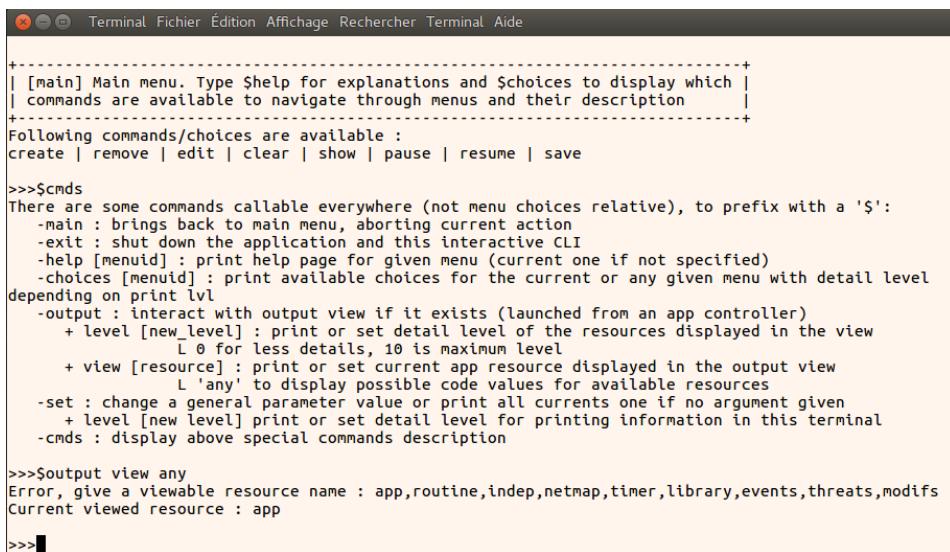
5.1 Justification des choix généraux

Comme l'application aspire à être déployée dans un environnement domestique, deux contraintes principales sont à prendre en compte pour son développement. La première est relative à l'utilisateur, qui de fait n'est pas une personne ayant une prédisposition particulière pour l'informatique. Un minimum de connaissances est quand même supposé : quelles sont les caractéristiques possibles d'un équipement sur le réseau, une idée de ce qu'est un scanner de réseau, de ce que sont la prise d'empreinte et l'analyse de trafic, etc. L'interface doit être assez simple pour pouvoir manipuler les modules et avoir une vue d'ensemble du réseau. La seconde contrainte concerne l'environnement restreint. Idéalement, l'application doit être fonctionnelle sur un équipement tel qu'une carte Raspberry PI, sans que l'interactivité avec l'application soit impactée. De plus, une utilisation dans un environnement non-graphique doit être possible, ce qui offrirait la possibilité de manipuler l'application à distance par le réseau, en utilisant *ssh* par exemple. Ce dernier point va de pair avec une utilisation sur une carte Raspberry qui est contrôlée à distance.

Afin de concilier ces deux contraintes, la première interface développée est en ligne de commande. Elle se divise en deux parties : l'interaction (CLI) qui se présente comme des menus contextuels et la vue où l'état actuel de l'application est affiché clairement. L'utilisateur n'interagit qu'avec la première, qui lui permet de contrôler l'application en arrière-plan et la vue. L'application se lance depuis un terminal avec possibilité de lui passer certains arguments, puis utilise ce terminal comme CLI pour afficher le contenu des menus et récupérer les commandes que l'utilisateur y écrit afin de les interpréter. La vue est quant à elle initialisée après le cœur logique de l'application et peut se présenter sous plusieurs formes en fonction des besoins de visualisation :

- **NoOutput** : aucune facilité n'est mise à disposition. Pour visualiser l'état actuel d'un élément de l'application, cela se fait via un menu *show* qui affiche la cible dans la CLI même.
- **PipeOutput** : l'élément cible à afficher ne l'est pas réellement, cette information textuelle est redirigée vers un pipe utilisable librement par l'utilisateur
- **ConsoleOutput** : apporte un support de visualisation à PipeOutput, invoquant un nouveau terminal du système qui va maintenir un affichage de l'information envoyée dans le pipe (il ne sert qu'à cela)

La CLI permet de sélectionner ce qui doit être affiché dans la vue, et avec quel degré de détail. Il peut s'agir de la vue d'ensemble de l'application, du statut courant d'un module dans la Queue et ses paramètres, etc. À titre d'exemple représentatif, la figure 13 montre ce à quoi ressemble l'interface et le système de menus.



The screenshot shows a terminal window with a dark background and light-colored text. At the top, there is a menu bar with options: Terminal, Fichier, Édition, Affichage, Rechercher, Terminal, and Aide. Below the menu, the terminal window displays the following content:

```
+-----+
| [main] Main menu. Type $help for explanations and $choices to display which |
| commands are available to navigate through menus and their description   |
+-----+
Following commands/choices are available :
create | remove | edit | clear | show | pause | resume | save

>>>$cmds
There are some commands callable everywhere (not menu choices relative), to prefix with a '$':
 -main : brings back to main menu, aborting current action
 -exit : shut down the application and this interactive CLI
 -help [menuid] : print help page for given menu (current one if not specified)
 -choices [menuid] : print available choices for the current or any given menu with detail level
 depending on print lvl
 -output : interact with output view if it exists (launched from an app controller)
   + level [new_level] : print or set detail level of the resources displayed in the view
     L 0 for less details, 10 is maximum level
   + view [resource] : print or set current app resource displayed in the output view
     L 'any' to display possible code values for available resources
 -set : change a general parameter value or print all currents one if no argument given
   + level [new level] print or set detail level for printing information in this terminal
 -cmds : display above special commands description

>>>$output view any
Error, give a viewable resource name : app,routine,indep,netmap,timer,library,events,threats,modifs
Current viewed resource : app
>>>■
```

FIGURE 13 – Interface CLI en mode ConsoleOutput

L'utilisateur peut entrer une valeur parmi les choix proposés, qui dépendent du menu actuel. Certaines commandes spéciales sont également invocables indépendamment, toutes préfixées par '\$'. Notamment la commande *\$output* qui permet de contrôler la vue, modifiant l'affichage courant.

5.2 Discussions des choix d'implémentation

5.2.1 Choix du langage

Le langage qui a été sélectionné pour implémenter le cœur de l'application est le Python dans sa dernière version (3.7). La première interface développée n'étant pas graphique, elle est également réalisée en Python. L'application ne demandant en soi aucune opération ou algorithme à forte complexité, un langage compilé n'apporterait pas d'amélioration significative en terme de performances par rapport au Python. En effet, les seules opérations potentiellement lourdes seraient celles effectuées par certains programmes sous-jacents des modules, mais c'est alors indépendant du langage choisi. Cependant, l'expérience de l'utilisateur n'en sera pas impactée, car de par le design choisi chacune de ces exécutions a lieu dans un fil dissocié de l'exécution principale.

Python possède également une librairie standard assez vaste que pour couvrir à peu près tous les besoins de l'application (voir détails des points suivants). Comme une interface complète en ligne de commande est un objectif en terme d'interactivité avec l'utilisateur, la manipulation simplifiée des flux standards et un formatage aisément des chaînes de caractères sont des atouts importants de Python.

5.2.2 Gestion du temps

Librairie utilisée : `time`

Le temps intervient dans la gestion de plusieurs éléments de l'application : timers d'expirations des modules actifs, lecture de la sortie à intervalle de temps donné pour les modules passifs, mise à jour de la vue. Cette multiplicité a motivé le choix de créer un objet gestionnaire du temps unique et global à l'application. Les autres éléments de l'application nécessitant une interactivité temporelle s'enregistrent auprès de cet objet **TimerThread** et doivent implémenter une interface **TimerInterface**. Cette interface ne contient que deux méthodes : *decr()* qui est l'action à effectuer et *is_decrementable()* qui renvoie un booléen à vrai si *decr()* doit être appelée.

Comme son nom l'indique, TimerThread va effectuer la gestion du temps dans un thread à part. Grâce aux primitives de la librairie `time`, le thread se réveille toutes les secondes pour notifier tous les objets qui se sont enregistrés, c-à-d appelle la méthode *decr()* de chacun individuellement si *is_decrementable()* de l'objet retourne vrai. L'action qui est ainsi effectuée par chacun des objets enregistrés chaque seconde est arbitraire et ne dépend que de l'implémentation qui en est faite dans l'objet.

5.2.3 Exécution des programmes sous-jacents

Librairie utilisée : `subprocess`

La librairie standard de Python permet la manipulation aisée de processus tiers sous forme d'objet. Le seul prérequis est le passage de la commande système destinée à appeler le programme, telle qu'elle serait utilisée en ligne de commande. Il est également possible de fournir et d'en récupérer les flux d'entrées et sorties standards. Ces facilités sont exploitées dans l'implémentation du schéma d'exécution pour les modules actifs et passifs.

5.2.4 Parsing des sorties

Comme les sorties des programmes sous-jacents sont spécifiques et hétérogènes, leur traduction vers les éléments de l'application n'est pas gérée automatiquement. C'est à la personne qui intègre le module d'écrire le code dédié à cette tâche. Cependant, des facilités sont envisageables pour cela. Actuellement, c'est le cas pour les sorties XML du programme Nmap : comme elles sont régies par une structure claire et définie quelqu'en soit l'utilisation, un parseur générique peut être fourni. Ce parseur peut être utilisé pour obtenir facilement des informations d'une sortie quelconque de Nmap et les manipuler comme des objets (correspondants à des *elements* dans le DOM¹⁷).

17. https://fr.wikipedia.org/wiki/Document_Object_Model

5.2.5 Stockage et agrégation des informations

Librairie utilisée : `lxml` (non intégré dans la librairie standard)

L'application maintient la Netmap, Routine, Librairie, etc. comme des objets, mais il est souhaitable que cette information soit retenue d'une exécution à l'autre et manipulable comme des fichiers indépendants. Considérant la quantité d'informations à stocker et la nature de conteneur de ces éléments, le choix a été fait d'utiliser le langage XML. Des méthodes de traduction ont donc été écrites pour les deux sens : des éléments de l'application vers une représentation XML et inversement. Cela a été fait pour les éléments suivants : Modules, Librairie, Routine, Netmap, Events. En terme de fichiers XML individuels, on en retrouve un pour la Librairie, un pour la Routine et un dernier pour la Netmap qui renseignent donc toute l'information des objets qu'ils contiennent :

- XML objet Librairie : une entrée (ie. un *element* pour le DOM) par module qui y est décrit, cette entrée contient des informations telles que l'id du module, le chemin du module python où est écrit la classe de l'objet Module, le nom de cette classe.
- XML objet Routine : divisé en deux parties pour le Panel et la Queue, remplit chacune d'entrée représentant les modules qui les peuplent. Chaque entrée renseigne l'id du module, son setid et ses paramètres courants au moment de la sauvegarde. Ces informations sont suffisantes pour réinstancier le Module à partir de la Librairie.
- XML objet Netmap : stocke une entrée par instance virtuelle, avec toutes les informations littérales qui lui étaient associées. Les Events liés à chaque IV spécifique sont également écrits pour ne pas être perdus.

Pour une exécution de l'application, on a donc trois fichiers principaux qui doivent être écrits si une sauvegarde non temporaire est souhaitée, et parsés au relancement. Il y a également d'autres informations diverses qui peuvent être stockées comme l'adresse mail de l'utilisateur. Afin de regrouper cette information (notamment les chemins des fichiers XML), un super-fichier de configuration globale de l'application est nécessaire. Pour manipuler les chemins de sauvegardes et ces configurations suite à des changements de préférences par l'utilisateur lors de l'exécution de l'application, un objet **CoreConfig** est utilisé.

Le cœur logique de l'application ne prend qu'un seul paramètre à son initialisation qui doit être une instance de CoreConfig, ce qui permet de séparer le chargement des ressources et leur instanciation de leur utilisation. Une instance de CoreConfig peut être créée en parsant le super-fichier de configuration, qui est un fichier au format YAML et contient notamment les chemins des autres fichiers XML cités précédemment. Les différents fichiers sont alors parsés et les objets correspondant instanciés dans un ordre prédéfini car il y a des dépendances entre eux. Si des chemins sont inexistant dans le fichier de configuration, les éléments sont instanciés vides et avec leurs paramètres par défaut. Plusieurs fichiers de configurations coexistant sont envisageables, par exemple si l'utilisateur souhaite utiliser des routines différentes en fonction du réseau dans lequel il lance l'application. La seule contrainte étant qu'à un lancement d'application un seul fichier de configuration est passé et potentiellement modifiable durant l'exécution.

5.2.6 Interactivité avec l'application

Le système de menus interactif a été conçu dans l'optique d'être utilisable simplement par une personne non initiée. Aucune librairie tierce n'a été utilisée pour le mettre en place, excepté les méthodes Python natives de prises d'entrées et complétion. Brièvement, un menu est représenté en interne par un dictionnaire dont plusieurs valeurs associées à des clés spécifiques doivent être définies. On y retrouve notamment :

- **desc** : une description textuelle du menu, quelle est sa fonction par rapport aux choix qu'il propose
- **choices** : une liste de choix qui peut être sous plusieurs forme (liste, dictionnaire, fonction qui retourne un des deux) qui sont les entrées possibles pour le menu courant
- **fct_choice** : une fonction dite de choix prenant une valeur en argument (qui sera le choix entré)

Une procédure bouclant sans fin permet de naviguer à travers les menus. Le menu courant correspond à un dictionnaire définissant les champs cités ci-dessus. Les choix disponibles sont calculés et mis en forme pour être présentés à l'utilisateur dont l'entrée est attendue. Une fois qu'il a fourni une entrée correcte, la fonction de choix est appelée sur cette entrée dont dépendent alors les actions effectuées. Par exemple, la fonction de transition de menu va simplement remplacer le menu courant par celui auquel correspond l'entrée de l'utilisateur, ce après quoi la procédure reprend du début.

5.2.7 Alertes et notifications par email

Librairies utilisées : [logging](#) et [smtplib](#)

L'utilisateur doit pouvoir être alerté efficacement quand une menace est détectée par un module. En l'occurrence, cette information doit être observable et récupérable depuis l'interface, mais également communiquée à l'utilisateur autrement puisqu'il n'est pas forcément devant. L'option la plus évidente est de notifier les menaces considérées comme importantes par mail. À cette fin, l'utilisateur peut fournir son adresse mail à l'application. Pour éviter des complications et la configuration d'un serveur dédié, le compte mail utilisé pour envoyer les notifications est celui de l'utilisateur lui-même. Il lui sera donc demandé son mot de passe afin de se connecter au serveur authentifié SMTP dont il utilise les services.

D'un point de vue implémentation, le lancement des alertes et la remontée des informations vers le centre d'événements de l'application depuis le code d'un Module doit être aisé. On distingue quatre façons de faire parvenir de l'information à l'utilisateur :

1. Par logging classique (utilisation des fonctionnalités natives de `logging`) avec les loggers `debug`, `info`, `error`, `cli`, `threats`, `modifs` qui redirigent l'information vers les flux d'entrées standards et les fichiers de logs.
2. En créant des objets Events qui seront gardés en mémoire dans l'application pour être manipulés. Facilité fournie : chaque objet Logger accessible classiquement en important `logging` (voir ci-dessus) est agrémenté d'une méthode `register_threat()/register_modif()` qui instancie cet Event, le garde en mémoire et le log de façon régulière en fonction du Logger concerné.
3. En enregistrant une entrée dans la zone de feedback de l'application, qui est un historique court affiché à l'utilisateur dans la vue (simples chaînes de caractères d'information). La méthode `log_feedback()` agrémenté le module `logging` à cet effet et permet également de logger de façon régulière l'information par un Logger désigné.
4. En envoyant un mail contenant un message personnalisé à l'utilisateur s'il a fourni une adresse. Cela se fait classiquement via le Logger `mail`.

Le module python `logging` a comme particularité de pouvoir être importé n'importe où dans le code mais de permettre l'accès aux même instances de Logger par un identifiant unique. Cela permet d'éviter de devoir transporter des références vers ceux-ci à travers le code. On peut donc utiliser cela et y coupler d'autres facilités personnalisées, en déclarant des nouvelles méthodes pour ces objets : `register_threat()/register_modif()` et `log_feedback()`.

De base, `logging` gère l'envoi de mail. Cependant, ce n'est pas le cas pour le SMTP authentifié qui est utilisé majoritairement. La librairie standard `smtplib` permet de pallier à ce problème en sous-classant l'utilitaire de base de `logging`.

6 Manuel d'utilisation

6.1 Prérequis et installation

Les prérequis généraux sont :

- Une distribution Linux (testé sous Debian, Ubuntu, Arch)
- Les programmes sous-jacents installés et accessibles en CLI, avec le nom renseigné par chaque attribut `cmd` d'un Module (du moins tous ceux utilisés dans la routine)
- Une version de Python ≥ 3.7
- Les droits de super-utilisateur pour lancer l'application

Au niveau logiciel, quelques dépendances doivent être installées si non présentes par défaut. Cela peut se faire avec la commande suivante :

```
sudo apt-get install python-yaml python3-lxml libxml2-dev libxslt1-dev xterm
```

Ensuite, une fois en possession des sources du projet (`git clone https://github.com/RemDec/IoTMonitor`), les dépendances aux librairies python et l'initialisation des fichiers requis peuvent être gérés en appelant à la racine

```
python3.7 setup.py develop
```

On distingue deux types d'utilisateurs qui sont susceptibles d'utiliser l'application, dans une optique différente. On a d'une part les utilisateurs normaux qui se servent de l'application pour son but premier : protéger leur réseau domestique. Leurs interactions se font uniquement depuis l'interface et avec les outils fournis. D'autre part, on a des personnes avec des connaissances plus avancées en programmation, capable d'assimiler le framework mis à disposition et intéressées par son utilisation. Ces dernières souhaitent intégrer leurs propres outils (sous-entendu programmes) pour pouvoir les manipuler dans l'application, et ont donc besoin d'une marche à suivre pour arriver à cette fin. Les deux cas d'utilisation sont distingués dans les sections suivantes qui font office de guide.

6.2 Utilisation classique

6.2.1 Point d'entrée de l'application

À la racine du projet se trouve le point d'entrée en ligne de commande de l'application `IoTMonitor.py`. Plusieurs arguments peuvent lui être passés, mais ils sont tous optionnels (l'application sera lancée avec des paramètres par défaut). En revanche, il est nécessaire d'invoquer la commande avec les droits de super-utilisateur et en utilisant une version de Python ≥ 3.7 . Une page d'aide est affichable avec le flag `--h`. Quelques autres paramètres sont très utiles :

- `--fileconfig FILEPATH` : indique le super-fichier de configuration tel que décrit au point 5.2.5.
- `--noautosave` : par défaut, quand l'application est quittée, l'état actuel des éléments est sauvegardé de sorte qu'au prochain lancement, leur état est restauré. Ce flag désactive cette fonctionnalité.
- `--noautoload` : par défaut, au lancement de l'application, l'état des éléments est restauré à celui qu'il était quand elle a été quittée (vierge si `noautosave` était mis). Désactive cette fonctionnalité, permettant de partir d'éléments vierges (routine, etc.).
- `--mail MAIL` : renseigne l'adresse mail de l'utilisateur que l'application utilisera pour les notifications d'alertes. Préférentiellement à utiliser avec le flag `--testmail` qui testera la validité et accessibilité de cette adresse en envoyant un email témoin.

Si une adresse mail est renseignée, le mot de passe pour le serveur SMTP authentifié le sera également, les mails étant envoyés à l'utilisateur avec sa propre adresse. Quand une menace est détectée par un module, celui-ci est susceptible d'en alerter l'utilisateur. Si aucune adresse mail n'est spécifiée, aucune tentative d'émission de mail n'aura lieu.

Pour lancer l'application dans une configuration recommandée pour débuter, la commande suivante peut être utilisée :

```
sudo python3.7 IoTmonitor.py -nal --mail untel@gmail.com --mserver smtp.gmail.com --testmail
```

Fournir le serveur SMTP n'est pas obligatoire, mais conseillé car sinon il sera deviné et donc peut être erroné. Cette commande va lancer l'application en mode ConsoleOutput, invoquant un nouveau terminal (xterm par défaut) qui servira de vue sur son état actuel. Le terminal depuis lequel elle est invoquée sera celui dans lequel les entrées de l'utilisateur sont récupérées. Deux remarques à prendre en compte pour cette étape :

1. Certain fournisseurs de services mail (Gmail par exemple) peuvent demander des autorisations supplémentaires. Voir les remarques dans la fiche d'aide.
2. Pour éviter certains bugs graphiques, il est préférable de lancer cette commande dans un terminal assez grand en terme de taille par rapport au reste de l'écran, voire en plein écran.

6.2.2 Navigation dans les menus

Une fois lancée, l'application initialise sa partie logique et l'utilisateur se retrouve dans la partie interactive, dont le point de départ est le menu principal. La combinaison de la vue et de l'interface devrait être similaire à celle illustrée dans la figure 14. Les conteneurs principaux, la Netmap et la Routine, y sont illustrés comme vides.

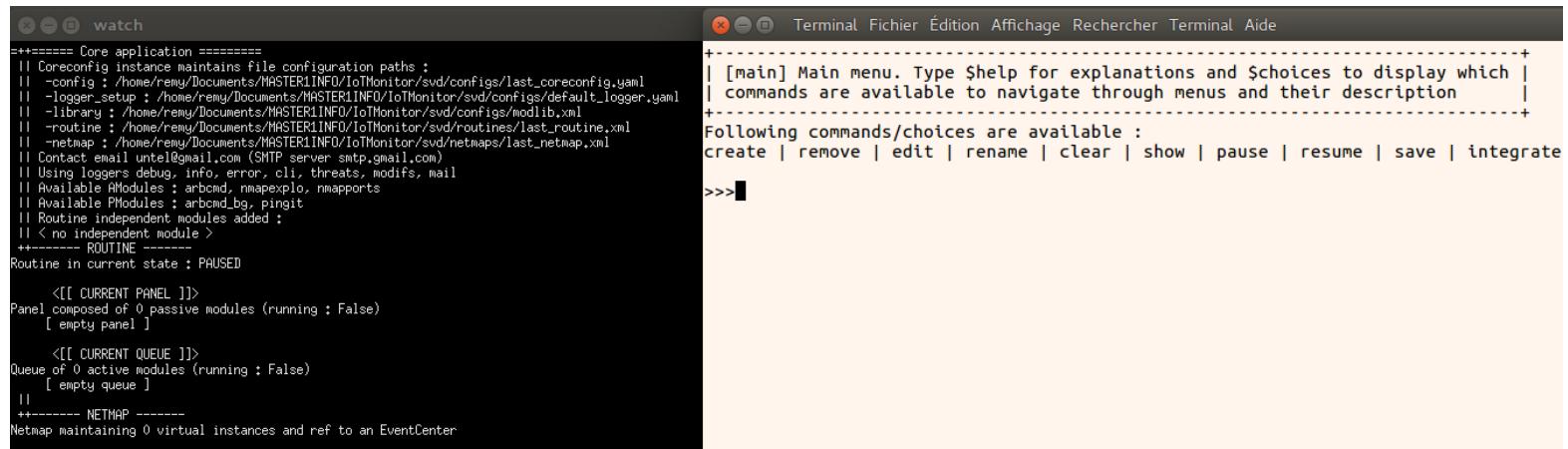


FIGURE 14 – Application au démarrage (fenêtre de vue et CLI)

Un cadre surplombant le champ de prise d'entrées renseigne quel est l'id du menu courant et une courte description de son utilité. Juste en dessous sont affichés les choix possibles pour ce menu, c'est-à-dire ce que l'utilisateur peut entrer comme valeur. Un système d'autocomplétion classique (touche **<Tabulation>**) permet de sélectionner rapidement un choix parmi ceux possibles, à valider par la touche entrée. De plus, appuyer sur entrée en n'ayant tapé que le début du mot le sélectionne automatiquement s'il n'y a aucune ambiguïté avec ce choix. Indépendamment du menu courant, des commandes spéciales sont invocables pour effectuer diverses actions, toutes à préfixer par le symbole '\$'. La liste de ces commandes est explicitée en tapant **\$cmds**.

Le menu principal permet d'accéder à toutes les fonctionnalités disponibles de l'interface, qui sont en fait utilisables depuis les autres sous-menus. Un appel à **\$choices** permet d'en savoir plus sur ces menus associés chacun à un choix, comme illustré par la figure 15. Pour revenir au menu principal depuis un menu quelconque, la commande **\$main** peut être utilisée.

```

Terminal Fichier Édition Affichage Rechercher Terminal Aide
>>>$choices
List of available choices/command for menu [main] :

create : Instantiate an object and integrate it in the application
remove : Remove an existing object in the app
edit : Edit an application element on the fly
rename : Rename an application element (changing its unique id)
clear : Clear application elements containers as netmap, routine, independent modules, etc.
show : Display current state of application resources
pause : Pause (interrupt running module threads) routine or its components
resume : Resume or start routine or its components
save : Save the current application configuration or components separately
integrate : Integrate a new written module in the application (library)

>>>■

```

FIGURE 15 – Description des sous-menus accessibles depuis le menu principal

6.2.3 Consultation de l'état des éléments

Bien que ce qui est affiché dans la vue soit contrôlable en utilisant la commande `$output`, il est parfois plus utile et précis d'afficher le statut de l'élément désiré dans le terminal de l'interface même. Cela peut se faire en se rendant dans le menu `show` accessible depuis le menu principal. Un panel d'éléments dont les détails sont consultables est proposé à l'utilisateur. La navigation dans plusieurs menus contextuels consécutifs peut être nécessaire pour spécifier l'élément visé : par exemple pour une instance virtuelle précise, il sera demandé d'entrer le mapid correspondant.

Lors de l'affichage de l'information désirée, un paramètre indiquant le niveau de détail est pris en compte. Pour modifier cette valeur avant l'affichage, la commande `set` peut être utilisée comme suit : `$set lvl [niveau]` où `niveau` est un entier entre 0 et 10, mais la majeure partie des éléments peuvent fournir plus de détail jusqu'au niveau 4. Il en va de même avec l'élément affiché dans la vue, la configuration passant cette fois par la commande `$output`.

6.2.4 Exemple pratique d'utilisation

Les actions rentrant en jeu pour mettre en place une surveillance du réseau domestique peuvent être réalisées aisément par n'importe quel utilisateur qui a compris les concepts abstraits de Modules et Routine. Cela peut se résumer à sélectionner quels modules instancier dans la Librairie parmi ceux disponibles par défaut et à les ajouter à la routine. La description des modules, le travail qu'ils effectuent et les paramètres qu'ils prennent sont accessibles en affichant la Librairie. Cela se fait donc en accédant au menu `show` depuis le menu principal, puis en sélectionnant l'élément Librairie en tapant `library`. Une description similaire à la figure 16 est affichée pour chaque module disponible.

```

< nmapexplo >
Module descriptor for [nmapexplo](A) def in modules.actives.nmapExplorer by AModNmapExplorer class.
Description : [nmapexplo] Nmap scan to discover hosts (-sn mode, no port scanning) by SYN/UDP probing on
common ports (need sudo)
Overlayer for program 'nmap -sn' with considered param setup :
| ~options (mandatory:F): other options compatible with -sn to pass to nmap
| -dflt prefixed val : [empty default value]
| ~SYNports (mandatory:T): Ports targeted for SYN probing
| -dflt prefixed val : -PS 21,22,23,80,443,3389
| ~UDPPorts (mandatory:T): Ports targeted for UDP probing
| -dflt prefixed val : -PU 53,135,137,161
| ~XMLfile (mandatory:T): The temp file where scan output will be written
| -dflt prefixed val : -oX /tmp/xml_nmapexplo.xml
| ~IP (mandatory:T): Target IP address(es) acceptable as Nmap syntax
| -dflt prefixed val : 192.168.0.3/24

```

FIGURE 16 – Affichage de la description pour le module `nmapexplo`

Cela renseigne que le programme sous-jacent est `nmap` et est d'office utilisé avec l'option `-sn` qui signifie "passer la phase de scan des ports" car comme l'indique la description fournie, il s'agit d'un module pour la découverte des hôtes uniquement. Viennent ensuite les paramètres qu'il peut prendre avec leur code, leur description textuelle et la valeur préfixée telle qu'elle sera insérée dans la ligne de commande appelant le programme. Un paramètre dit "mandatory" sera toujours présent dans cette ligne de commande et prendra la valeur par défaut associée si aucune autre n'est fournie. Les valeurs par défaut des paramètres optionnels (non "mandatory") ne sont donc juste là qu'à titre indicatif, car le paramètre sera ignoré si aucune valeur ne lui est manuellement fournie. À noter que certaines valeur par défaut de paramètres peuvent être recalculées dynamiquement à l'instanciation du module, comme l'IP dans ce cas pour

correspondre à celle du réseau local.

Une fois la sélection des modules à utiliser terminée, il faut les instancier dans la Routine. Cela se fait en revenant au menu principal avec `$main` et choisissant le menu `create`. Le choix par défaut est indiqué par `< . >`, qui est bien ici sur `module`, on peut donc le sélectionner directement en appuyant sur Entrée. La liste des ids des modules disponibles dans la Librairie est affichée, en premier lieu les modules passifs et à la ligne suivantes les actifs. Après avoir entré l'id voulu, une série de questions amenant à des sous-menus contextuels sont posées afin de configurer l'instance du module et son comportement dans la Routine (figure 17) :

1. Utiliser les paramètres par défaut ?
 - Si oui, le module verra ses paramètres courants mis à leur valeur par défaut et aucun paramètre optionnel ne sera utilisé.
 - Si non, pour chaque paramètre existant une valeur sera demandée, appuyer sur Entrée revenant à utiliser celle par défaut.
2. Ajouter l'instance dans la routine ?
 - Si oui, l'instance du module sera ajoutée dans le conteneur correspondant à ton archétype (Panel ou Queue).
 - Si non, l'instance est ajoutée à l'ensemble des modules indépendants, mais il est déconseillé de l'utiliser au stade actuel de développement.
3. Fournir des instances virtuelles spécifiques auxquelles l'exécution du module devrait se rapporter ?
 - Si oui, un sous-menu permettra de donner les mapids des IVs concernées. Le module pourra utiliser cette information à des fins diverses, vraisemblablement il exécutera son programme sous-jacent vis-à-vis de celles-ci mais cela dépend entièrement de son implémentation.
 - Si non, le module s'exécutera indifféremment où en considérant toutes les VIIs s'il avait besoin de cibles données.
4. Spécifier un timer ? (intervalle de lecture pour un module passif ou d'expiration pour un actif)
 - Si un entier est entré, utilisation de cet entier comme le nombre de secondes
 - Si rien n'est spécifié, utilisation de la valeur par défaut
5. Spécifier un setid utilisé pour identifier l'instance du module dans la Routine ?
 - Si une chaîne de caractères est entrée le setid sera mis à sa valeur
 - Si rien n'est spécifié, utilisation de l'id du module

```
+-----+  
| [newMod] Choose a module in the current library and instantiate it |  
+-----+  
Following commands/choices are available :  
arbcmd | nmapexplo | nmappers  
arbcmd_bg | pingit  
  
[mod_id] :nmapexplo  
[nmapexplo] use defaults params (Y/n)? n  
[nmapexplo] append it in routine (Y/n) ?  
Parameter options (optional, no prefix) : other options compatible with -sn to pass to nmap  
[default:<empty>] :  
Parameter SYNports (mandatory, flag -PS) : Ports targeted for SYN probing  
[default:21,22,23,80,443,3389] :80  
Parameter UDPports (mandatory, flag -PU) : Ports targeted for UDP probing  
[default:53,135,137,161] :  
Parameter XMLfile (mandatory, flag -oX ) : The temp file where scan output will be written  
[default:/tmp/xml_nmapexplo.xml] :  
Parameter IP (mandatory, no prefix) : Target IP address(es) acceptable as Nmap syntax  
[default:192.168.0.3/24] :192.168.0.*  
  
Indicate specific VIIs the [nmapexplo] execution should be relative to ? (y/N) :  
  
Set queue expiration interval timer if desired (numeric) or enter to pass  
[default:60] :20  
  
Give a setid if desired (alphanumeric)  
[setid] :nameinroutine|
```

FIGURE 17 – Instanciation du module `nmapexplo` à placer dans la Routine

Après remplissage de la Routine avec plusieurs instances de modules différents, on peut observer son état dans la vue (figure 18). Pour chaque instance de module, on a un résumé de son état actuel informant également du nombre de thread(s) courant(s) qui lui sont associés. Sur la figure 18 on observe qu'aucun thread n'est enregistré donc aucun programme sous-jacent n'est en cours d'exécution. C'est normal car la Routine est toujours en état de pause. Pour lancer la routine automatique de surveillance, le menu **resume** doit être accédé depuis le menu principal et **entire routine** doit y être choisi.

```
+----- ROUTINE -----
Routine in current state : PAUSED

<[[ CURRENT PANEL ]]>
Panel composed of 1 passive modules (running : False)
+-----+
| pingit [pingit] ~ BG[0/0] COMM[0/0 ~ 30s] !
+-----+

<[[ CURRENT QUEUE ]]>
Queue of 2 active modules (running : False)
+-----+
| nameinroutine 20s [20][nmapexplo] ~ Thlist[0/0] !=>| nmapports 60s [60][nmapports] ~ Thlist[0/0] !
+-----+
```

FIGURE 18 – État de la Routine après remplissage

La Routine illustrée permet de découvrir et analyser les équipements du réseau domestique pour en regrouper les caractéristiques. La Netmap devrait donc se remplir au fur et à mesure des exécutions des modules pour converger vers une représentation du réseau avec les différents équipements qui le peuplent. À titre d'exemple, la Netmap de la figure 19 est celle obtenue après un court moment.

```
+----- NETMAP -----
Netmap maintaining 7 virtual instances and ref to an EventCenter
+-----+ +-----+ +-----+ +-----+
| device0 [?] 0 /!\ 10 -o- | | device1 [?] 0 /!\ 7 -o- | | device2 [?] 0 /!\ 1 -o- | | device3 [?] 0 /!\ 10 -o-
| ====== | | ====== | | ====== | | ======
| MAC: A0:21:B7:52:81:69 | | MAC: E4:58:E7:3E:C4:C4 | | MAC: 08:EE:8B:D7:75:AB | | MAC: C4:9D:ED:92:DE:DE
| IP: 192.168.0.1 | | IP: 192.168.0.2 | | IP: 192.168.0.4 | | IP: 192.168.0.5
| Hostname:< empty > | | Hostname:< empty > | | Hostname:< empty > | | Hostname:< empty >
| Ports table: | | Empty ports table | | Empty ports table | | Ports table:
| | 23 : < telnet, TCP, closed | | manufacturer: Samsung Electr | | manufacturer: Samsung Electr | | 135 : < msrpc, TCP, open >
| | 80 : < http, TCP, open > | +-----+ +-----+ +-----+ | | 139 : < netbios-ssn, TCP, o
| | 8080 : < http-proxy, TCP, c | | manufacturer: Netgear | | manufacturer: Netgear | | 445 : < microsoft-ds, TCP,
| | manufacturer: Netgear | +-----+ +-----+ +-----+ | |
+-----+ +-----+ +-----+ +-----+
| device4 [?] 0 /!\ 10 -o- | | device5 [?] 0 /!\ 9 -o- | | device6 [?] 0 /!\ 8 -o- |
| ====== | | ====== | | ====== |
| MAC: < empty > | | MAC: 14:32:D1:77:CA:F9 | | MAC: 70:4F:57:EC:95:E3 |
| IP: 192.168.0.3 | | IP: 192.168.0.6 | | IP: 192.168.0.25
| Hostname:< empty > | | Hostname:< empty > | | Hostname:< empty >
| Ports table: | | Empty ports table | | Ports table:
| | 80 : < http, TCP, open > | | manufacturer: Samsung Electr | | 22 : < ssh, TCP, open >
| | manufacturer: Samsung Electr | +-----+ +-----+ | | 80 : < http, TCP, open >
+-----+ +-----+ +-----+ +-----+
```

FIGURE 19 – État de la Netmap après que la routine de la figure 18 ait tourné un court moment

Cette représentation de la Netmap a pour but de donner une vue d'ensemble du réseau. Dans l'en-tête de chaque vignette d'IV, le symbole entre crochet représente l'état de connexion au réseau ('?' indique un flou), *n* / !\ indique le nombre de menaces trouvées pour l'équipement et *n* -o- le nombre de modifications de ses champs. Pour consulter en détail ces informations, le choix **virtual instance** du menu **show** est utilisable.

Pour agrémenter et améliorer la représentation du réseau, l'utilisateur peut rentrer ou corriger lui-même les valeurs des champs d'une IV, voire la créer de toute pièce. Cela se fait via le menu **edit** et **create** respectivement. Il en va de même pour les instances de Module. D'autres menus gèrent des fonctionnalités diverses, tel que **save** qui permet d'écrire les fichiers de sauvegarde de la Routine et de la Netmap décrits au point 5.2.5, ainsi que de redéfinir les références vers les fichiers de sauvegardes à utiliser à l'initialisation définies dans le super-fichier de configuration.

6.3 Utilisation du point de vue du programmeur

6.3.1 Écrire un module

À un Module correspond, en terme de code, une classe astreinte à suivre le framework imposé. Par convention, le nom de celle-ci doit commencer par "AMod" pour un Module actif et "PMod" pour un Module passif. En fonction de l'archétype associable au programme sous-jacent à abstraire, cette classe doit également hériter de la classe abstraite correspondante : *ActiveModule* ou *PassiveModule*. De fait, elle devra également en implémenter les méthodes abstraites. Sans ses conditions, l'intégration d'un nouveau module à l'application ne se fera pas correctement.

Deux options sont possibles pour l'implémentation d'une classe correspondant à un nouveau Module. La plus simple des deux est aussi celle qui offre le moins de contrôle : il s'agit de deux sous-classes (abstraite également) à *ActiveModule* et *PassiveModule* implémentant l'utilisation des facilités fournies par ces dernières (voir sous-section 4.3). Ces deux sous-classes sont respectivement *FacilityActiveModule* et *FacilityPassiveModule*. Elles implémentent le maximum de ce qui peut l'être en ne laissant comme abstraites que les méthodes spécifiques à chaque Module (i.e. liée avec le programme sous-jacent). L'autre option, qui nécessite plus de travail mais permet de personnaliser davantage le comportement d'un Module, est de simplement hériter de *ActiveModule* ou *PassiveModule* en implémentant toutes les méthodes abstraites. La possibilité d'utiliser les facilités fournies de ces deux classes est alors laissée comme libre.

Plus concrètement, les méthodes abstraites à implémenter dans une sous-classe de *FacilityActiveModule* et *FacilityPassiveModule* sont les suivantes :

- *get_module_id()*, *get_cmd()*, *get_description()* qui sont des chaînes de caractères pour respectivement l'id unique du Module, la commande système utilisée pour appeler le programme en CLI et une description textuelle du module
- *get_scheme_param()* qui sert à décrire quels paramètres le module peut prendre et de quelle façon il sont traités : un dictionnaire dont les clés sont les noms des paramètres et les valeurs des triplets (*valeur par défaut, obligatoire, préfixe en CLI*)
- *get_desc_params()* retourne un dictionnaire mappant chaque nom de paramètre avec une description textuelle de son utilité
- *build_final_cmd()* crée la commande telle qu'elle serait passée en CLI à partir de *get_cmd()*, les valeurs courantes des paramètres et le schéma de traitement de ces valeurs donné par *get_scheme_param()*.
- *parse_output(output)* est la méthode de parsing de la sortie du programme qui est censée faire la traduction vers des changements dans le reste de l'application

Au final, implémenter un nouveau module avec ces facilités revient à écrire une classe d'à peu près 150 lignes dont la moitié constituent la fonction de parsing des sorties, seul travail laborieux mais incontournable.

Afin de quand même faciliter le travail à effectuer pour l'implémentation en sous-classant directement *ActiveModule* ou *PassiveModule*, deux classes "squelette" types déjà structurées sont fournies. Il suffit d'en reprendre le code en y apportant les personnalisations voulues et les informations à remplir (id du Module, etc.).

Des facilités de logging et de création d'Events sont également utilisables. Elle permettent de simplement déclarer de nouvelles modifications de valeurs des champs d'instance virtuelle, des menaces les concernant, envoyer un mail, etc. Pour rappel, plusieurs loggers sont présents de base à l'initialisation, avec des objectifs différents : afficher un message dans la CLI interactive, enregistrer les événements importants dans des fichiers de logs y étant dédiés, et d'autres utilitaires.

6.3.2 Intégrer un module dans l'application

Pour intégrer un nouveau Module à l'environnement afin qu'il y soit manipulable, il faut l'ajouter à la Library qui est chargée à l'initialisation. Pour l'instant, cette librairie est représentée par un fichier XML unique dont le chemin est `svd/configs/modlib.xml`, celui-ci n'étant pas modifiable pour pointer vers un autre fichier. La Library est construite en le parsant, chaque entrée s'y trouvant représente un Module et ses informations. Afin de faire le lien avec le code python du Module, le nom du package et le nom du module python dans lequel la classe est écrite sont fournis dans les attributs XML de l'entrée. Un exemple d'entrée dans ce document est le suivant :

```
<actmod modid="nmapexplo" cmd="nmap -sn" pymod="modules.actives.nmapExplorer" pyclass="AModNmapExplorer">
  <desc>[nmapexplo] Nmap scan to discover hosts (-sn mode, no port scanning) by SYN/UDP probing ...</desc>
  <defparams>
    <param code="options" mandatory="False" default="" prefix="">other options compatible with -sn to ...</param>
    <param code="SYNports" mandatory="True" default="21,22,23,80,443,3389" prefix="-PS">Ports targeted for ...</param>
    <param code="UDPPorts" mandatory="True" default="53,135,137,161" prefix="-PU">Ports targeted for UDP ...</param>
    <param code="XMLfile" mandatory="True" default="/tmp/xml_nmapexplo.xml" prefix="-oX ">The temp ...</param>
    <param code="IP" mandatory="True" default="192.168.0.3/24" prefix="">Target IP address(es) acceptable ...</param>
  </defparams>
  <savedparams/>
  <dependencies/>
</actmod>
```

Trois options s'offrent au programmeur pour intégrer son module, qui sont par ordre de facilité :

1. Intégration via l'interface CLI (menu `integrate`) en fournissant le module python
2. Par programme, en instanciant un petit objet utilitaire `ModuleIntegrator` ou l'objet de contrôle de la librairie entière
3. Par écriture directe dans le fichier `modlib.xml`

L'intégration par CLI fait l'hypothèse que le code décrivant l'objet Module se trouve au bon endroit dans les package pythons destinés à les regrouper par archétype `modules.passives` et `modules.actives`. Le code doit contenir une classe respectant les conditions évoquées au point 6.3.1 et qui est le code du Module. Supposons que le fichier contenant le code s'appelle `mymodule.py`, alors celui-ci correspond au module python `mymodule` (dans l'arborescence `modules.passives.mymodule` ou `modules.actives.mymodule`). Arrivé dans le menu `integrate`, le programmeur renseigne l'archétype de son nouveau Module. Il lui est en suite demandé d'entrer le nom du module python dans lequel il est défini : en l'occurrence `mymodule`. Si toutes les conventions ont été respectées, une entrée devrait être écrite dans le fichier `modlib.xml`, celui-ci reparsé et le Module devrait apparaître parmi ceux disponibles.

L'objet utilitaire pour intégrer un Module à partir d'une de ses instances (donc créée par programme ou dans l'interpréteur Python) est trouvable dans le package `src.utils.filesManager`. Il s'agit de la classe `ModuleIntegrator`, qui prend en paramètre de constructeur une instance du Module à intégrer ou une chaîne de caractères désignant le module python à importer pour trouver la classe correspondante (`modules.passives.mymodule`). En instanciant un objet `ModManager` du module `src.utils.moduleManager` qui correspond à ce qu'on désignait par Library, il est également possible d'intégrer la nouvelle entrée au fichier grâce à la méthode `add_to_modlib_file(mod instance)`.

La troisième façon d'intégrer un Module nouvellement écrit est celle la moins recommandée. Il s'agit simplement d'écrire l'entrée XML manuellement. Cela devrait être fait dans le fichier qui est chargé d'office : `modlib.xml`. La DTD pour ce fichier XML n'ayant pas encore été écrite, il est vivement recommandé d'éviter cette méthode et d'utiliser une des deux autres méthodes qui écrivent l'entrée automatiquement.

7 Tests et évaluation de l'application

7.1 Évaluation de la détection des équipements

7.2 Évaluation de la détection de menaces

7.2.1 Simulation d'équipements virtuels

7.2.2 Utilisation dans un réseau domestique peuplé

7.3 Déploiement sur Raspberry Pi et utilisation distante

Le but est de pouvoir utiliser et manipuler l'application tournant sur une carte Raspberry Pi distante par ssh, donc en mode non-graphique. Cela doit pouvoir se faire sans perdre de fonctionnalités par rapport à une utilisation de l'application en mode ConsoleOutput. Le mode à considérer ici est PipeOutput.

Avant de s'y essayer, il est nécessaire d'installer Python 3.7 depuis les sources car Raspbian ne dispose pas encore de cette version. On utilise les instructions proposées dans cet article : <https://github.com/instabot-py/instabot.py/wiki/Installing-Python-3.7-on-Raspberry-Pi>. Il est également nécessaire d'installer les programmes sous-jacents (*nmap*, *p0f*, etc.), l'application faisant l'hypothèse qu'ils le sont. Ce après quoi on applique les instructions d'installation données au début de cette section. Ces opérations étant toutes réalisables via une connexion ssh existante, on suppose qu'il est possible d'en établir une seconde depuis un autre terminal (qui servira de vue). Dans la première qui servira d'interface interactive on lance donc l'application avec :

```
sudo python3.7 IoTmonitor.py -lvl 3 -m outpiped
```

Un pipe est alors créé automatiquement dans /tmp et alimenté toute les secondes avec l'état actuel de l'application. Pour observer ces modifications de façon continue, on peut alors utiliser la commande **watch** dans le second terminal :

```
watch -t -n 0,5 cat output_monitor
```

Comme attendu, l'application est alors utilisable de la même façon que en mode ConsoleOutput et offre des fonctionnalités identiques.

8 Conclusion

La première partie de ce projet a été l'occasion de se plonger dans ce que la littérature proposait en terme de cybersécurité pour les foyers. Beaucoup de thèmes ont été abordés en surface car l'application à réaliser n'était pas fixée. Aussi, une vue globale était nécessaire pour décider quels seraient les tenants et les aboutissants de la partie implémentation.

Pour cette application, le soin apporté à l'intégration générique de n'importe quel programme arbitraire est un investissement sur le long terme. À l'heure actuelle, le nombre de module disponibles est restreint par manque de temps non pas pour leur implémentation, mais pour la recherche et les expérimentations nécessaires avant l'intégration. Effectivement, la seule difficulté est inhérente au fait qu'il est nécessaire de traduire les sorties de chaque programme spécifique. Outre cela, beaucoup d'améliorations sont encore facilement apportables à l'application : interface graphique, gestion des dépendances, etc.

Références

- [1] Application firewall. Wikipedia article, URL : https://en.wikipedia.org/wiki/Application_firewall.
- [2] Cable broadband technology gigabit evolution. Dernière lecture le 27/11/2018.
- [3] Cisco asa. Wikipedia article, URL : <https://en.wikipedia.org/wiki/Cisco ASA>, dernière lecture le 19/12/2018.
- [4] Firewall(computing). Wikipedia article, URL : [https://en.wikipedia.org/wiki/Firewall_\(computing\)](https://en.wikipedia.org/wiki/Firewall_(computing)).
- [5] Internet growth statistics par internet world stats. <http://www.ti.com/lit/ml/swrb028/swrb028.pdf>. Dernière lecture le 26/11/2018.
- [6] Les firewalls. Community article, URL : <https://www.frameip.com/firewall/>.
- [7] Pare-feu(informatique). Wikipedia article, URL : [https://fr.wikipedia.org/wiki/Pare-feu_\(informatique\)](https://fr.wikipedia.org/wiki/Pare-feu_(informatique)).
- [8] Security in the Internet of Things. https://www.windriver.com/whitepapers/security-in-the-internet-of-things/wr_security-in-the-internet-of-things.pdf, 2015.
- [9] Alhafidh Basman M.Hasan and William Allen. Design and simulation of a smart home managed by an intelligent self-adaptive system. *Int. Journal of Engineering Research and Application*, 2016.
- [10] Isha Batra and Kr. Luhach Ashish. Analysis of lightweight cryptographic solutions for Internet of Things. *Indian Journal of Science and Technology*, 9, July 2016.
- [11] Dan-Radu Berte. Defining the IoT. *De Gruyter*, 2018.
- [12] Cisco. *CLI Book 2 : Cisco ASA Series Firewall CLI Configuration Guide*, 9.6, May 2018. URLs : <https://www.cisco.com/c/en/us/td/docs/security/asa/asa96/configuration/firewall/asa-96-firewall-config/access-idfw.html>, <https://www.cisco.com/c/en/us/td/docs/security/asa/asa96/configuration/firewall/asa-96-firewall-config/access-idfw.pdf>.
- [13] Jorge Granjal, Edmundo Monteiro, and Jorge Sá Silva. Security for the internet of things : A survey of existing protocols and open research issues. *IEEE Communications Surveys & Tutorials*, pages 1–1, 07 2015.
- [14] Robert J. Shimonski, Debra Littlejohn Shinder, and Thomas W. Shinder, editors. *The Best Damn Firewall Book Period*. Syngress, Burlington, 2003.
- [15] Kalle Kokko. Next-generation firewall case study. Master's thesis, South-Eastern Finland University of Applied Sciences, 2017.
- [16] Sylvain Kubler, Kary Främling, and Andrea Buda. A standardized approach to deal with firewall and mobility policies in the IoT. *Pervasive Mobile Computing*, 2014.
- [17] Vishwas Lakkundi and Keval Singh. Lightweight dtls implementation in coap-based internet of things. 09 2014.
- [18] Engin Leloglu. A review of security concerns in Internet of Things. *Journal of Computer and Communications*, 2017.
- [19] Manuel Leos Rivas. Securing the home IoT network. SANS Institute Reading Room, URL : <https://www.sans.org/reading-room/whitepapers/hsoffice/paper/37717>, 2014.
- [20] Huichen Lin and Neil Bergmann. IoT privacy and security challenges for Smart Home environments. *Information (online MDPI journal)*, 2016.
- [21] Imran Makhdoom, Mehran Abolhasan, Justin Lipman, Ren Liu, and Wei Ni. Anatomy of threats to the internet of things. *IEEE Communications Surveys and Tutorials*, PP, 10 2018.
- [22] Norbert Nthala and Ivan Flechais. Rethinking home network security. *European Workshop on Usable Security*, 2018.
- [23] Dariusz Palka and Marek Zachara. Learning web application firewall - benefits and caveats. pages 295–308, 01 2011.
- [24] Alaauddin Shieha. *Application Layer Firewall Using OpenFlow*. PhD thesis, University of Colorado, 2014.
- [25] Wikipedia contributors. Horus scenario, exploiting a weak spot in the power grid. Dernière lecture le 30/01/2019.
- [26] Wikipedia contributors. 6lowpan — Wikipedia, the free encyclopedia. <https://en.wikipedia.org/w/index.php?title=6LowPAN&oldid=899429432>, 2019. Dernière lecture le 01/02/2019.
- [27] Wikipedia contributors. Mirai (malware) — Wikipedia, the free encyclopedia. [https://en.wikipedia.org/w/index.php?title=Mirai_\(malware\)&oldid=886619081](https://en.wikipedia.org/w/index.php?title=Mirai_(malware)&oldid=886619081), 2019. Dernière lecture le 7 mars 2019.