Les risques d'OpenFlow et du SDN

Maxence Tury
maxence.tury@ssi.gouv.fr

ANSSI

Résumé Par opposition aux réseaux traditionnels, le paradigme *Software-Defined Networking* prône une séparation des fonctions de routage et de celles de transfert de paquets. L'échange d'information entre ces deux plans passe par le développement de nouveaux protocoles et outils dont la sécurité n'a été éprouvée que par encore peu d'études. Ces technologies suscitent beaucoup d'intérêt mais leur déploiement expose les réseaux à des risques méconnus. Cet article présente les spécificités du standard ouvert OpenFlow en mettant l'accent sur certaines de ses faiblesses. La mise en place d'un écosystème SDN complet et les failles observées au sein d'implémentations récentes font l'objet de mises en garde supplémentaires.

1 Introduction

1.1 Le paradigme *Software-Defined Networking*

L'industrie des réseaux se développe depuis une vingtaine d'années sur la base d'un modèle où les routeurs assurent à la fois le contrôle de la topologie du réseau (c'est-à-dire l'accomplissement des fonctions de routage) et le transfert des paquets. La gestion de la topologie s'accompagne souvent d'une charge de calcul significative. Par exemple, le protocole de routage OSPF exige de chaque routeur un travail important chaque fois qu'une modification topologique du réseau lui est signalée. Ce calcul est en effet nécessaire pour appliquer l'algorithme de Dijkstra, qui permet de déterminer les routes optimales vers les hôtes distants.

Un paradigme alternatif, cependant, attire actuellement l'attention d'un nombre croissant d'acteurs du marché. Le SDN [1] se caractérise par la séparation physique du plan de contrôle et du plan de données, ainsi que la centralisation des fonctions de contrôle. Administration mise à part, le fonctionnement d'un routeur traditionnel est reproduit grâce à deux types d'outil distincts. Un contrôleur, d'une part, est chargé du calcul et de la mise à jour des tables de routage de plusieurs équipements, d'autre part. Ces dispositifs, que l'on appellera abusivement *switchs*, sont responsables

1. *Software-Defined Networking*

du transfert des paquets d'une interface à une autre en fonction des règles établies par le contrôleur. OpenFlow est une des solutions développées pour permettre la communication entre contrôleurs et *switchs*.

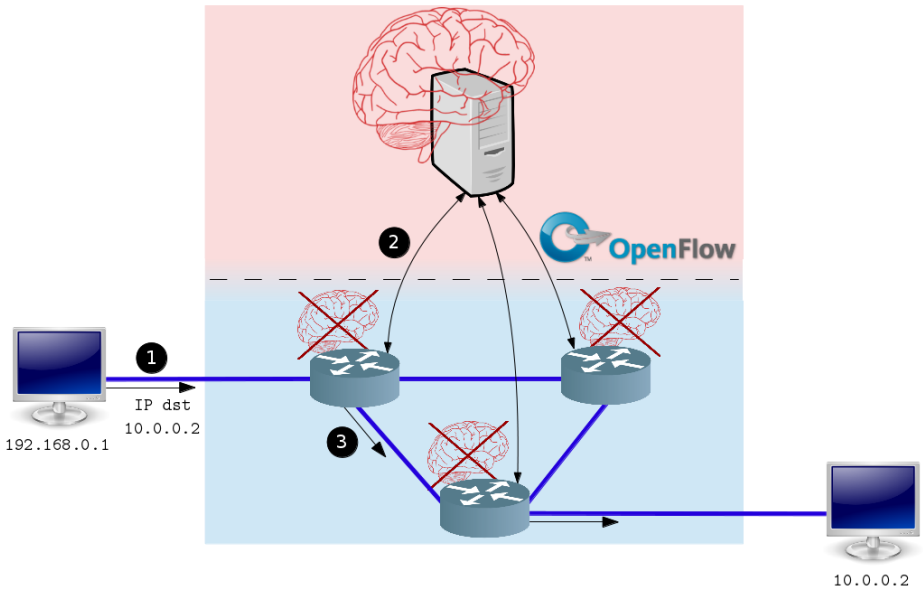

Figure 1. Schéma d'un réseau SDN simple

Dans une approche SDN, la charge de calcul associée au contrôle est en grande partie retirée des routeurs. La figure 1 illustre le routage d'un paquet entre deux réseaux distants via une implémentation de SDN avec le protocole OpenFlow. Les routeurs discriminent les paquets et déterminent leur interface de sortie, mais ce comportement découle de règles émises par le contrôleur. Le routeur qui réceptionne le paquet de l'envoi (1) signale l'évènement au contrôleur et reçoit en retour des règles au cours d'un échange (2), afin de décider du transfert (3) vers le routeur suivant approprié. Ce comportement est dit réactif ; un comportement proactif consisterait en la transmission de règles avant que le routeur ne reçoive les paquets associés. Les routeurs effectuent ainsi essentiellement des fonctions de commutation—d'où la désignation de « *switchs* OpenFlow ».

Dans un souci de clarification, un seul contrôleur figure sur le schéma, mais il est envisageable et conseillé, principalement pour des questions de

résilience, d'intégrer au réseau d'autres contrôleurs qui pourront prendre le relais en cas d'incident.

1.2 Historique

Bien que des formulations des concepts sous-jacents puissent être retracés au début des années 2000, la désignation SDN est récente, précédée de peu par les premiers travaux sur OpenFlow en 2008 [9]. La version 1.0.0 des spécifications du protocole, destinée à la production, était publiée début 2010. Le pont avec l'industrie a rapidement été franchi, en 2011, avec la création de l'Open Networking Foundation par Deutsche Telekom, Facebook, Google, Microsoft, Verizon et Yahoo. L'ONF est l'organisme principal encourageant à l'adoption des pratiques SDN, et encadre l'évolution du protocole OpenFlow.

En 2012, Google a présenté son installation pionnière : une architecture SDN qui effectue désormais le routage de leur backbone utilisateur ainsi que de leur backbone interne, mise en place en deux ans [5]. Les premières observations mettaient notamment en évidence l'utilité de la centralisation des mécanismes de contrôle de bande passante, de perte de trames, ou encore de différenciation d'applications. Le modèle SDN aurait ainsi facilité la gestion de la qualité de service, et permis une amélioration de la tolérance aux incidents. D'autres groupes ont plus tard annoncé avoir implémenté OpenFlow, parmi lesquels Amazon, Microsoft ou encore AT&T. L'adoption semble à ce jour limitée aux datacenters et aux WAN les plus vastes. Malgré ces déploiements, très peu de données quantitatives sur les performances des installations SDN ont été rendues publiques.

La plupart des équipements réseau qui prennent en charge OpenFlow sont aussi capables d'effectuer du routage traditionnel, ce qui autorise une transition progressive vers un déploiement SDN. NEC, HP ou encore Brocade produisent ces *switchs* OpenFlow hybrides depuis 2011. Par ailleurs, la communauté travaille sur la compatibilité de *switchs* virtuels tels qu'Open vSwitch, avec une forte contribution de VMware.

Il faut noter qu'OpenFlow n'est pas l'unique implémentation des concepts SDN. Par exemple, le système ONE de Cisco exploite le protocole OpFlex et propose une approche intégrée, depuis la couche matérielle jusqu'aux interfaces d'administration.

1.3 État de l'art

Bien que certains développeurs envisagent déjà des solutions de déploiement SDN à des fins de consolidation de la sécurité des réseaux, la

sécurité des implémentations en général et d'OpenFlow en particulier font encore l'objet de peu d'études. Les publications *OpenFlow Vulnerability Assessment* [3] et *OpenFlow : A Security Analysis* [6], proposent d'effectuer une analyse exhaustive des faiblesses du protocole. Les résultats rapportés dans le présent document mettent plus fortement l'accent sur la faisabilité des attaques et le rôle des différentes implémentations, dans l'esprit de *Floodlight DDoS Vulnerability* [7] et *A denial of service attack against the Open Floodlight SDN controller* [4], études datées de fin 2013.

2 Aperçu du protocole OpenFlow : le traitement par flux

Les versions les plus couramment implémentées du protocole sur les *switchs* OpenFlow actuels correspondent aux spécifications 1.0 et 1.3. Bien qu'elles ne soient pas compatibles entre elles, la 1.3 peut être envisagée dans sa globalité comme une extension de la 1.0. La description et les tests qui suivent portent donc sur la 1.3, sauf mention contraire.

OpenFlow sert de lien entre le plan de contrôle et le plan de données. L'échange de messages se fait au cours d'une session TCP établie via le port 6653 du serveur contrôleur. Le comportement d'un *switch* OpenFlow est alors déterminé par une ou plusieurs tables de flux (*flow table*). Chaque table s'assimile à un ensemble de flux, qui consistent chacun en une liste de discriminants relatifs au contenu d'une trame, associée à des actions de traitement à appliquer aux trames correspondantes. Lorsqu'un paquet parvient au *switch*, les valeurs contenues dans ses en-têtes sont comparées aux différents jeux de valeurs enregistrés dans la première table de flux du *switch*. Les actions qui peuvent suivre sont de diverses natures : transfert de la trame par une interface, suppression, modification de champs d'en-têtes, et/ou transmission du traitement à une table de flux ultérieure.

Le regroupement des flux définit l'intégralité des fonctions qui pourront être appliquées aux paquets reçus par le *switch*. Un flux par défaut doit être défini pour chaque table, explicitant le comportement à adopter lorsqu'une trame ne correspond à aucun flux plus spécifique. Un cas courant consiste à envoyer au contrôleur un message de type `PACKET_IN`. Selon le paramétrage, celui-ci peut contenir l'intégralité de la trame, un nombre limité de ses octets (possiblement aucun) ou encore une référence de mémoire tampon. Le contrôleur répondra généralement par un `PACKET_OUT` donnant l'instruction à suivre.

En mettant en relation une adresse de destination précise, contenue dans l'en-tête des paquets IP, avec une des interfaces en particulier, il est possible d'établir une règle de routage classique. Les spécifications

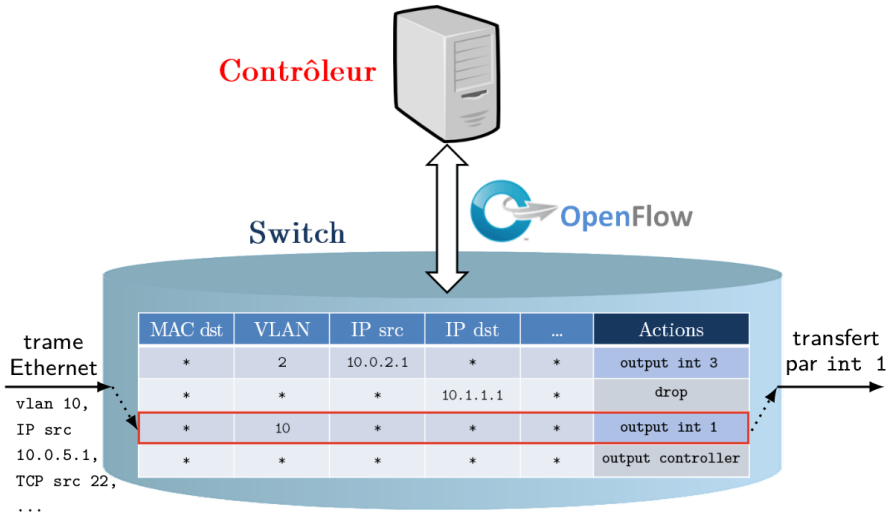

Figure 2. Schématisation du modèle par flux OpenFlow

prennent en charge plusieurs types de champs d'en-tête : adresses MAC, EtherType, identifiants de VLAN, adresses IPv4 et IPv6, ports TCP et UDP, labels MPLS, etc. La figure 2 illustre le traitement d'une trame parvenant à un *switch* OpenFlow contenant une seule table de flux : parmi les différents champs d'en-têtes de la trame, l'appartenance au VLAN 10 la fait correspondre à un des flux du *switch*, dont l'action associée consiste en un transfert par l'interface 1.

À des fins de disponibilité, un *switch* peut entretenir des sessions OpenFlow avec différents contrôleurs. Pour chacune des connexions, le contrôleur peut adopter un rôle d'esclave ou de maître. Un esclave est informé de certains changements de statut, par exemple de l'état des interfaces, mais ne peut pas agir sur la configuration ; un maître possède tous les droits (comme le rôle par défaut) mais il ne peut en exister qu'un seul. Ce rôle peut être redéfini auprès du *switch* à tout moment.

Dans le cas où le *switch* n'est plus en mesure de joindre aucun des contrôleurs auxquels il est lié, deux comportements sont définis : le mode *fail secure*, où toutes les actions d'envoi de paquet au contrôleur sont ignorées, mais les flux déjà présents persistent, ainsi que le mode *fail standalone*, où les paquets sont tous traités avec les fonctions de routage traditionnelles (si le *switch* est hybride).

3 Faiblesses du protocole

3.1 Perméabilité de la liaison contrôleur–*switch*

Du point de vue de la sécurité, une différence majeure du modèle SDN avec les installations traditionnelles est la centralisation du contrôle. Le contrôleur est un point critique du réseau (d'autant plus si aucun contrôleur auxiliaire n'a été déployé), et il doit être l'unique source des consignes OpenFlow envoyées aux *switchs*. Si l'envoi de celles-ci a lieu sur le réseau de production, ce qui n'est pas découragé par les spécifications, la sécurité de l'installation repose alors sur l'authentification des communications. Cependant, la prise en charge de TLS exigée pour OpenFlow 1.0 a été rendue optionnelle à partir d'OpenFlow 1.1 : « *The OpenFlow channel is usually encrypted using TLS, but may be run directly over TCP.* » [8]

Jusqu'à mi-2014, très peu de *switchs* et aucun contrôleur libre n'assuraient ce service. Plusieurs implémentations TLS ont été publiées depuis, qui devront être privilégiées. En effet, en l'absence de mécanisme d'authentification, la centralisation au niveau du contrôleur opérée par OpenFlow permet à tout attaquant ayant gagné accès au réseau d'administration d'agir à tout moment sur le paramétrage du *switch* et de configurer le réseau de production comme souhaité. Il est aussi envisageable pour un attaquant de se placer en *man-in-the-middle* afin de filtrer les commandes d'un administrateur légitime ainsi que les remontées d'information des *switchs*. Enfin, l'écoute passive de messages OpenFlow non chiffrés permet d'obtenir de nombreux renseignements sur les réseaux maintenus par le contrôleur.

La protection d'un *switch* OpenFlow passe donc par le **cloisonnement des réseaux d'administration et de production**, ainsi que l'**utilisation exclusive de TLS sur le réseau d'administration**. Le cloisonnement et l'intégrité des communications sont des pratiques établies et disponibles sur des protocoles de routage historiques tels que BGP et OSPF, et restent valables pour OpenFlow.

3.2 Risques de déni de service côté *switch*

Comme avec les *switchs* traditionnels, il est possible de provoquer un DoS sur un *switch* OpenFlow en le soumettant à un trafic soutenu. Un *switch* incapable de traiter l'ensemble des paquets qui lui parviennent en supprimera une partie, si ce n'est l'intégralité, parfois silencieusement [9]. Bien que l'attaque ne soit pas propre au protocole, il est important de noter que plusieurs modèles de *switchs* OpenFlow physiques traitent

certaines actions au niveau logiciel et non matériel : les débits nécessaires pour provoquer un déni de service seront alors bien moindres.

Une première approche propre au protocole vise à surcharger la mémoire tampon du *switch* (s'il en dispose) avec des paquets en attente d'être envoyés au contrôleur. La mémoire pourrait saturer pour deux raisons principales : la réception intensive de paquets à envoyer au contrôleur, et l'absence de réponse du contrôleur aux `PACKET_IN`. Il n'existe pas de message d'erreur signalant au contrôleur cette saturation. Les spécifications ne permettent pas de vider la mémoire tampon du *switch* de façon indépendante (par exemple en supprimant les trames non traitées au bout d'un certain délai), mais elles proposent si besoin de désactiver l'utilisation de la mémoire tampon—au risque de transférer trop de données et de surcharger le contrôleur. Par ailleurs, OpenFlow 1.0 exige l'envoi de `PACKET_IN` pour chaque paquet ne correspondant à aucun flux connu, mais OpenFlow 1.3 permet à l'administrateur de définir les actions à appliquer à de tels paquets, notamment leur suppression.

En supposant la possibilité d'envoyer des messages OpenFlow au contrôleur, un attaquant pourrait aussi chercher à surcharger une table de flux d'un *switch*. Dans le cas d'une table pleine, la version 1.3 des spécifications prévoit simplement d'envoyer un message d'erreur à un contrôleur qui tenterait d'ajouter une nouvelle règle : si l'administrateur n'intervient pas pour supprimer certains flux, aucune nouvelle règle ne sera acceptée par le *switch*. Seule la version suivante du protocole répond à ces problèmes avec les mécanismes d'expulsion et d'alerte de taux d'occupation d'une table. Reste à établir une politique d'expulsion appropriée, et évaluer pour chaque *switch* le taux d'occupation à partir duquel les performances seront notablement dégradées. Par ailleurs, les implémentations du protocole sur certains *switchs* sont mises à mal par une forte occupation des tables de flux, comme présenté dans la section 4.3.

3.3 Risques de déni de service côté contrôleur

Dans la mesure où les ressources en mémoire du contrôleur sont susceptibles d'être supérieures à celles du *switch* de plusieurs ordres de grandeur, surcharger la base de flux du contrôleur est une méthode potentiellement coûteuse pour l'attaquant. Ce modèle d'attaque présuppose la possibilité d'enregistrer des flux sur le contrôleur.

Un risque plus important se présente par rapport au système de maître–esclaves. Sans mécanisme d'authentification, il est simple pour un contrôleur adverse d'usurper le rôle de maître, ou du moins d'empêcher qu'un contrôleur légitime qui souhaite être maître puisse paramétrer le

switch. En effet, quand bien même ce contrôleur légitime effectuerait une requête de rôle à chaque fois qu'il serait averti de son passage au statut d'esclave, il suffirait à l'adversaire d'adopter exactement le même comportement pour empêcher le contrôleur légitime d'effectuer des actions de configuration. Bien que les changements de droits n'y soient toujours pas restreints, la version 1.4 définit des messages d'avertissement qu'un *switch* devra envoyer à un esclave qui vient d'être démis de son rôle de maître.

4 Implémentations sur le *switch* de test

L'examen du protocole n'est pas suffisant pour évaluer la sécurité d'une installation OpenFlow : il faut aussi en étudier les implémentations. Nous avons donc choisi de mener plusieurs expérimentations sur des *switchs* de test. La visée de cet article n'étant pas d'évaluer la qualité d'un produit mais de faire état des enjeux de sécurité actuels associés au protocole OpenFlow, la marque des *switchs* utilisés n'est pas révélée. Nous sommes entrés en contact avec l'équipementier concerné pour lui présenter nos résultats et résoudre les problèmes qui n'étaient pas encore traités dans les versions de *firmware* les plus récentes.

Les premières observations ont été faites à l'aide du *firmware* v1, qui prend en charge la version 1.0 d'OpenFlow, mais qui n'assure pas les fonctions développées dans les versions ultérieures du protocole. Au cours de l'étude, un *firmware* v2 a été publié, ajoutant la prise en charge d'OpenFlow 1.3. La version utilisée est précisée pour chaque test.

Le contrôleur *open source* Floodlight a d'abord été utilisé pour envoyer des messages OpenFlow aux *switchs*. Les commandes d'administration passent d'abord par l'envoi de requêtes HTTP à Floodlight, qui se charge ensuite de l'envoi des messages OpenFlow. Cependant l'API de Floodlight ne couvre pas l'écriture de tous les messages définis dans les spécifications, et formate parfois les réponses reçues sans expliciter chacun des champs. Afin de pallier ces manques deux modules OpenFlow ont été développés pour Scapy [1], pour les versions 1.0 puis 1.3, qui sont accessibles depuis les sources du projet. En complément, un contrôleur rudimentaire a été développé pour la gestion des sessions OpenFlow, dont le code figure en annexe A, accompagné d'un exemple de client en B.

4.1 Écarts par rapport aux spécifications

Bien que les spécifications prennent en charge la gestion de différents VLAN, les implémentations testées définissent des réseaux de productions,

ou « instances » OpenFlow, qui sont chacun associés à un seul VLAN défini par l'administrateur à l'initiation du déploiement.

Prise en charge de TLS. Parmi les autres écarts annoncés par la documentation du *firmware* v1 figure l'absence de prise en charge des paquets IP fragmentés et, plus notablement, celle de TLS. Avec le *firmware* v2, le constructeur annonce cette prise en charge. À ce stade, nous n'avons cependant pas été en mesure d'établir manuellement une session TLS entre un *switch* et un serveur `openssl` accompagné de notre script contrôleur. Nous avons contacté l'équipementier à propos des caractéristiques nécessaires à un contrôleur en vue d'exploiter la prise en charge de TLS.

Écarts aux spécifications non annoncés. Chaque réponse OpenFlow doit contenir dans son en-tête un identifiant de transaction identique à celui utilisé pour la requête. S'il est vrai que chaque réponse envoyée par le *switch* comprend un identifiant identique à celui qui figurait dans la requête, il faut cependant noter que le *switch* accepte n'importe quel identifiant lorsque le contrôleur répond à un `HELLO` (pour initier la session) ou à un `ECHO_REQUEST` (pour maintenir la session ouverte). Tant que les implémentations ne cherchent pas à exploiter cet identifiant au-delà de la facilité d'appairage entre requêtes et réponses qu'il est censé apporter, cet écart aux spécifications ne devrait pas porter à conséquence. En l'absence de définition par les spécifications d'un comportement à suivre, ce choix devrait être documenté par l'équipementier. De façon annexe, les spécifications requièrent qu'un `FEATURES_REQUEST` soit envoyé par le contrôleur en début de session, mais les *switchs* testés ne l'exigent pas.

Mécanisme de gestion de rôles. Après réception d'un `ROLE_REQUEST`, un *switch* est censé répondre avec un `ROLE_REPLY` qui contient en en-tête la même valeur de `generation_id` que celle présente dans le premier message. Pour les *switchs* testés, cette valeur est toujours 0. De plus, la comparaison des `generation_id` utilisée pour valider les requêtes de maître est faussée. La distance définie entre les `generation_id` est « circulaire », de sorte que, sur 4 octets, par exemple toute valeur entre `0` et `0x7fffffff` est censée être considérée plus grande que `0xffffffff`. Or sur le modèle testé, `0xffffffff` est une valeur absolument maximale, et toute requête de rôle de maître ou d'esclave dans l'intervalle de valeur précédent sera refusée. En fait, si le `generation_id` a atteint `0xffffffff` il n'est plus possible d'effectuer les prochaines requêtes autrement qu'avec la même valeur de `0xffffffff`. (Les spécifications sont ambigües quant à la validité de telles

requêtes car il n'est pas dit que la nouvelle valeur devrait être *strictement* supérieure à la précédente ; cette implémentation semble contrevenir à l'esprit de la définition mais permet au moins de conserver la fonctionnalité des changements de rôle.) De ce fait il suffit qu'une requête soit effectuée avec `0xffffffff` pour prévenir toute incrémentation et rendre inutile ce mécanisme de gestion des maîtres.

4.2 Concurrence de flux

Pour un paquet qui correspond à deux flux, la version 1.0 des spécifications d'OpenFlow utilisait un attribut de priorité qui déterminait laquelle des deux listes d'actions appliquer. Nous appelons « concurrence » la situation dans laquelle ces flux sont de priorité identique. En cas de concurrence, les spécifications indiquent que le flux à appliquer est un choix d'implémentation. Afin d'établir les choix suivis par les *switchs* testés, nous avons utilisé trois flux de priorités identiques dont les conditions de correspondance sont décrites dans le tableau 1.

	interface d'entrée	ethertype	adresse IP source	autres attributs
champs 1	1	*	*	*
champs 2	1	0x0800	*	*
champs 3	1	0x0800	192.168.10.3	*

Table 1. Champs de correspondance des 3 flux

Le tableau 2 présente les flux privilégiés en fonction de la version de *firmware* utilisée, ainsi que du triplet d'actions associées aux flux retenus. Ces actions consistaient, soit en le transfert du paquet vers une interface prédéfinie (représenté par un **F**, pour *Forward*), soit en la suppression du paquet (**D** pour *Drop*). Les champs grisés montrent laquelle des trois actions a été appliquée à un paquet reçu via l'interface 1 et originaire de 192.168.10.3 (quand l'action appliquée est associée à plusieurs flux lors d'un même test, les compteurs maintenus par le module OpenFlow du *switch* permettent d'établir lequel des flux a été appliqué).

Contrairement à l'intuition selon laquelle le *switch* choisirait d'appliquer les flux les plus précis, le flux 3 n'est jamais appliqué. Les flux privilégiés ne sont pas non plus les plus génériques, comme en témoigne le fait que le flux 1 n'est pas toujours appliqué. Les tests effectués avec le

	v1		v2			
action 1	F	D	F	D	F	D
action 2	D	F	D	F	F	D
action 3	F	D	F	D	F	D

Table 2. Actions privilégiées par le switch de test

firmware v1 laissent penser que la sélection dépend des champs de correspondance et non des actions associées, mais ce jugement n'est plus valable avec le *firmware* v2. En l'absence d'indications de la part du constructeur, il est difficile d'établir les règles de ce processus d'élection.

Afin de prévenir toute action découlant d'une concurrence involontaire, une solution consiste à utiliser le drapeau `CHECK_OVERLAP` dans chaque requête d'ajout de flux. L'activation de ce drapeau permet au *switch* de rejeter une demande d'ajout de flux avec un message d'erreur si celui-ci rentre en concurrence avec un flux déjà enregistré. Les modules Scapy ont permis de vérifier que cette option était correctement implémentée, mais elle reste inaccessible depuis l'API proposée par le contrôleur Floodlight.

4.3 Performances en fonction des flux

Nous étudions ici certaines réponses en performance d'un *switch* sous *firmware* v1 en fonction des flux qui y sont enregistrés dans la table d'une instance OpenFlow 1.0. Ces résultats s'appuient sur des relevés quantitatifs effectués à l'aide de l'outil `iperf`. Ceux-ci n'ont qu'un intérêt relatif et non absolu : ils varient en fonction de nombreux paramètres de flux (en plus de dépendre des modèles de *switch*). La plupart des résultats restent valables en utilisant la table d'une instance OpenFlow 1.0 de v2, ou bien la première des deux tables modifiables d'une instance OpenFlow 1.3.

Recherche de flux dans une table. On pourrait supposer que, lorsqu'un *switch* cherche à savoir si une trame appartient à un flux pour lequel il possède des consignes de traitement, le parcours de la table de flux du *switch* se fait par ordre de priorité décroissante et s'arrête une fois qu'un flux correspondant a été trouvé (vu qu'au plus un seul flux doit s'appliquer). Les observations laissent cependant penser que la recherche n'est pas optimisée de cette façon. En effet, lorsqu'on fait appliquer deux flux de priorité maximale (qui établissent un simple transfert d'une interface à une autre), les performances seront notablement moins bonnes si la table contient quelques milliers d'autres flux de priorité faible ou nulle. Quand

bien même les priorités étaient prises en compte pour parcourir la table de flux, le *switch* est susceptible d'utiliser d'autres critères de recherche qui auront un impact sur les performances. La section traitant des flux concurrents témoigne de l'influence des champs de correspondance et des actions sur l'application d'un flux. Avec le *firmware* v2, on peut aussi observer qu'il est assez simple de remplir une table de flux en faisant varier l'attribut EtherType ; mais en faisant varier d'autres champs (par exemple le port TCP source), le *switch* est susceptible de redémarrer de manière impromptue après l'enregistrement de quelques milliers de flux, ce qui s'accompagne de la perte de tous les flux enregistrés jusque-là.

Limite de la taille d'une table. Lorsqu'on envoie un `FEATURES_REQUEST` au *switch* sous v1, celui-ci répond avec un nombre d'entrées maximal de $1048576 = 2^{20}$. La table atteint cependant sa capacité maximale avec $65536 = 2^{16}$ flux, et toute tentative ultérieure d'ajout de flux provoque un message d'avertissement. Cet écart semble s'expliquer par la façon non standardisée dont les flux sont enregistrés dans les *switchs* (à partir de la version 1.2 les spécifications précisent d'ailleurs « *a flow table entry might consume more than one table entry, depending on its match parameters* »). En dépit de variations apportées aux attributs des flux, chacun semble occuper 2^4 entrées de la table de flux des *switchs* sous v1. Quoi qu'il en soit, les versions ultérieures du firmware sollicitées par un `FEATURES_REQUEST` renvoient une valeur de 2^{16} qui reflète directement les capacités du *switch*.

Indisponibilité à saturation. Le serveur est en mesure de générer des flux en temps constant à envoyer au *switch*. Une instruction transmise à l'interface d'administration du *switch* permet d'afficher le nombre de flux réellement enregistrés. Le temps de réponse de la CLI augmente au fur et à mesure de l'accroissement de la table de flux du *switch*. Par contre, les performances de transfert du *switch* (en fonction des flux) ne s'en ressentent pas tant que la table contient moins de dix mille flux. Passé ce seuil, le *switch* peut rompre la connexion OpenFlow avec le contrôleur et/ou redémarrer, ce qui est parfois susceptible de provoquer un déni de service au bout de moins de cinq minutes d'envoi de flux. Plus rarement, le contrôle du *switch* peut être perdu et un redémarrage physique être nécessaire. Cette rupture peut aussi être déclenchée par une sollicitation du CLI, ou même par un ping. La figure 3 représente les délais de réponse à 200 pings traversant le *switch*, envoyés chacun à une seconde d'intervalle. Le profil linéaire décroissant montre que le *switch* est muet pendant des

périodes de 16-17 secondes avant de renvoyer un ensemble de réponses quasi-simultanément. Les délais minimaux correspondent à des périodes de réponses nominales, de l'ordre de la dizaine de millisecondes. Enfin les relevés répétés d'une ou deux réponses avec un délai de 4-5 secondes restent à ce stade inexpliqués.

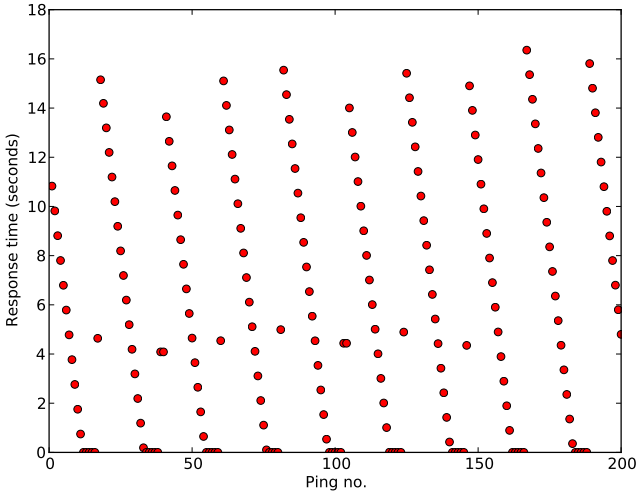

Figure 3. Temps de réponse pour chaque ping

4.4 Recherches dans la table de flux

Sous certaines conditions, les fonctions de recherche de flux à partir du CLI exhibent un comportement anormal, pouvant mener à un redémarrage indésirable de l'équipement et à la perte des flux enregistrés jusque-là. Les tests présentés ci-dessous ont été réalisés avec le *firmware* v2 et la version 1.0 du protocole. Les résultats avec le *firmware* v1 (forcément OpenFlow 1.0) sont qualitativement identiques ; par contre les recherches effectuées avec le *firmware* v2 sur une instance OpenFlow 1.3 fonctionnent correctement. Cette erreur concernant les instances OpenFlow 1.0 semble avoir été corrigée avec le *firmware* v3.

Nous avons d'abord transmis deux flux au *switch*, qui vont supprimer d'une part toutes les trames ARP, d'autre part tous les segments TCP. Une première recherche sur tous les flux du *switch* renvoie correctement les deux

flux présents. Ensuite, la recherche de tous les flux qui s'appliquent aux trames ARP renvoie comme attendu le premier flux enregistré. Par contre, la recherche de tous les flux qui s'appliquent aux segments TCP renvoie en double le second flux enregistré. Enfin, la dernière recherche sur les flux s'appliquant aux trames LLDP, qui ne devrait rien renvoyer, provoque le redémarrage du *switch*. Après deux à trois minutes d'interruption de service, il est possible d'y réaccéder en SSH. On constate par ailleurs que tous les flux ont été supprimés.

Nous avons reproduit ces résultats (flux en double et redémarrage intempestif) plusieurs fois, avec des champs d'en-tête distincts, et de différentes valeurs. Le déclenchement du redémarrage ne semble pas non plus dépendre de l'ordre d'enregistrement des flux. La configuration la plus simple consiste à enregistrer un flux sans restriction (pour OpenFlow 1.0, cela signifie des valeurs de champs toutes nulles et des *wildcards* tous activés) et un autre flux avec un attribut non nul. Une recherche sur le champ et la valeur particulière est susceptible de renvoyer deux fois le second flux. A quelques exceptions près, les autres recherches provoqueront le redémarrage du *switch*.

5 Implémentation sur un contrôleur de test

Les premiers tests ont été effectués avec le contrôleur Floodlight, par l'intermédiaire de son API SFP [2]. Il s'agit d'une API de type REST : l'administrateur envoie des requêtes HTTP au contrôleur, qui envoie ensuite les messages OpenFlow appropriés aux *switchs* concernés. Par exemple, avec Floodlight v0.90, une requête `POST` sur `http://<control>/wm/staticflowentrypusher/clear/<switch>/json` provoque au niveau du contrôleur une recherche de tous les flux gérés par le module SFP. Floodlight envoie ensuite au *switch* ciblé des messages OpenFlow de type `FLOW_MOD` et de code `OFPFC_DELETE`, dont les attributs correspondent aux flux identifiés. Au cours des tests, un défaut d'implémentation a été détecté, qui permet à un attaquant ayant accès à l'API de masquer des flux aux administrateurs légitimes.

Le principal problème est lié à l'utilisation par le module SFP de noms fournis par l'administrateur en tant que clés primaires au sein de sa base locale de flux. Lorsqu'un flux est soumis au module pour insertion dans la base, en cas de conflit avec le nom d'un flux déjà présent, c'est le flux le plus ancien qui est supprimé. En règle générale, le contrôleur se charge de transmettre la commande de suppression du flux en question,

2. Interface de programmation Static Flow Pusher [2]

mais si l'ancien et le nouveau flux sont relatifs à des *switchs* différents, le contrôleur oubliait de supprimer l'ancien flux.

Le scénario consiste alors pour l'attaquant à d'abord envoyer son flux malveillant `fl1` sur le *switch* `sw1` par l'intermédiaire de SFP, avec un nom quelconque. Ensuite, il réutilise ce nom pour indexer un flux quelconque `fl2`, envoyé sur un second switch `sw2`. Dans la base de flux entretenue par SFP, les attributs de `fl1` sont écrasés par ceux de `fl2`. Il suffit ensuite de supprimer `fl2` pour revenir à la base de flux initiale, bien qu'un flux supplémentaire `fl1` soit désormais présent sur `sw1`.

Dans la mesure où le module SFP a été développé de sorte qu'il n'interagisse pas avec les flux gérés par d'autres modules de Floodlight, lorsqu'une requête comme celle donnée en exemple est effectuée, SFP s'appuie sur sa base de flux et non sur les flux présents sur le *switch*. Aussi, les destinataires des `FLOW_MOD` peuvent ne pas correspondre avec l'ensemble des flux présents sur le *switch*, et dans l'état correspondant à la fin du scénario précédent, le flux `fl1` ne serait pas supprimé.

Le module StaticFlowPusher reste capable de lister tous les flux présents sur un *switch* avec d'autres commandes, mais, par design, ne cherchera pas à synchroniser sa base avec la liste obtenue. Le problème a été remonté aux développeurs du contrôleur, et le test omis a été patché dans Floodlight v1.0. Cette version permet aussi l'utilisation de HTTPS pour les requêtes transmises à l'API, forcerait a priori l'authentification des administrateurs souhaitant agir sur les *switchs* via l'API.

6 Conclusion

Les architectures *Software-Defined Networking* modifient le paysage de l'industrie des réseaux. Il est difficile d'affirmer si les technologies associées vont s'implanter durablement parmi les produits liés aux réseaux, mais il est nécessaire d'étudier leur sécurité avant qu'ils n'émergent, et le cas échéant de guider leur développement selon des pratiques saines.

Bien que le protocole soit fonctionnel, nous avons présenté dans cet article plusieurs éléments des spécifications d'OpenFlow qui sont susceptibles de porter atteinte à la sécurité d'un réseau. Le SDN diffère des paradigmes réseaux classiques, ce qui introduit de nouveaux enjeux de sécurité aux côtés de pratiques essentielles (bien que non explicitées dans les normes) telles que l'utilisation d'un réseau d'administration dédié, la mise en place d'une solution d'authentification et d'intégrité telle que TLS, ou encore la redondance des équipements en charge du routage.

Nous avons ensuite mis en évidence plusieurs failles d'implémentations d'OpenFlow sur un modèle de *switch*. Nous insistons sur le fait que les anomalies relevées sont propres à un modèle particulier, et que les plus graves d'entre elles ont spontanément été résolues par l'équipementier dans les versions les plus récentes du *firmware*, ne laissant essentiellement que certains soucis de performance en cas de table de flux saturée. D'autre part, bien que les tests effectués pour cet article portent sur le produit d'un seul équipementier, cela ne préjuge en rien de la confiance à accorder au reste des produits de cet équipementier, ni aux *switchs* OpenFlow d'autres équipementiers qui n'ont pas été examinés. Ces résultats suffisent cependant à montrer que les administrateurs réseau souhaitant développer des prototypes innovants doivent avoir conscience du manque de robustesse et du caractère imprévisible de certaines technologies SDN encore immatures. Ce jugement dépasse le cadre des *switchs*, comme illustré avec le défaut corrigé dans le contrôleur Floodlight.

L'usage d'OpenFlow, allié à la mise en œuvre d'autres études pratiques de sécurité, sera nécessaire au standard pour bénéficier des retours nécessaires à l'affermissement de ses spécifications et des outils associés. Dans l'attente de ces améliorations, il faut considérer que les éventuels bénéfices de performance associés à un déploiement du protocole OpenFlow en production se feraient au détriment de la sécurité générale du réseau, par rapport à l'utilisation de protocoles de routage historiques conformément à des principes de protection connus.

Références

1. Scapy. `http://bb.secdev.org/scapy/`.
2. Static Flow Pusher API – Floodlight Controller. `http://www.openflowhub.org/display/floodlightcontroller/Floodlight+REST+API`.
3. K. Benton, L.J. Camp, and C. Small. OpenFlow Vulnerability Assessment. `http://conferences.sigcomm.org/sigcomm/2013/papers/hotsdn/p151.pdf`, August 2013.
4. J.M. Dover. A denial of service attack against the Open Floodlight SDN controller. `http://dovernetworks.com/wp-content/uploads/2014/03/OpenFloodlight-03052014.pdf`, December 2013.
5. Urs Hoelzle. OpenFlow @ Google. `http://www.youtube.com/watch?v=VLHJUfgxEO4`, May 2012.
6. R. Klöti. Openflow : A security analysis. `ftp://ftp.tik.ee.ethz.ch/pub/students/2012-HS/MA-2012-20.pdf`, April 2013.
7. N. Solomon and Y. Francis and L. Eitan. Floodlight DDoS Vulnerability. `http://www.slideshare.net/YoavFrancis/floodlight-openflow-ddos`, September 2013.
8. OpenFlow Switch Specification. `https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.1.0.pdf`, February 2011. Rev. 1.1.0.

9. OpenFlow Switch Specification. https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.4.0.pdf, October 2013. Rev. 1.4.0.

A Serveur contrôleur OpenFlow 1.0 et 1.3

```
#!/usr/bin/python
from select import select
import getopt, signal, socket, struct, sys
from scapy.contrib import openflow as OF
from scapy.contrib import openflow3 as OF_04
from scapy.all import *

class ServerOF(object):

    def __init__(self, portN, portS):
        self.outputs = []
        self.serverN = socket.socket(socket.AF_INET, socket.
            SOCK_STREAM)
        self.serverN.setsockopt(socket.SOL_SOCKET, socket.
            SO_REUSEADDR, 1)
        print 'Binding North on port {0}...'.format(portN)
        self.serverN.bind(('', portN))
        self.serverN.listen(5)
        self.serverS = socket.socket(socket.AF_INET, socket.
            SOCK_STREAM)
        self.serverS.setsockopt(socket.SOL_SOCKET, socket.
            SO_REUSEADDR, 1)
        print 'Binding South on port {0}...'.format(portS)
        self.serverS.bind(('', portS))
        self.serverS.listen(5)
        self.verb = False
        signal.signal(signal.SIGINT, self.sighandler)

    def sighandler(self, signum, frame):
        for o in self.outputs:
            o.close()
        self.serverS.close()
        self.serverN.close()

    def display(self, data, msg_type, cls):
        try:
            msg = cls.get(map(ord, msg_type)[0])(data)
            msg.show()
        except:
            print '\nError during message parsing!\n'
            hexdump(data)
        print '\n'

    def process_sw_msg(self, data, s):
        if data[0] != '\x01' and data[0] != '\x04':
            print 'Received wire protocol is neither OFv1.0 nor OFv1
                .3.\n'
            hexdump(data)
```

```
            print data
        else:
            mod = OF if data[0] == '\x01' else OF_04
            if not self.verb and data[1] != '\x02':
                mod.OpenFlow(None, data)(data).show()
                print ''

            if data[1] == '\x00':
                s.send(str(mod.OFPTHello()))
            elif data[1] == '\x02':
                s.send(str(mod.OFPTEchoReply()))


    def serve(self):
        inputs = [self.serverS, self.serverN]
        self.outputs = []
        admins = []
        running = 1

        while running:
            inputready, outputready, exceptready = select.select(
                inputs, self.outputs, [])

            for s in inputready:

                if s == self.serverS:
                    self.sw, address = self.serverS.accept()
                    print 'Switch connection {0} from {1}'.format(
                        self.sw, address)
                    inputs.append(self.sw)
                    self.outputs.append(self.sw)

                elif s == self.serverN:
                    admin, address = self.serverN.accept()
                    print 'Connection {0} from {1}'.format(admin,
                        address)
                    inputs.append(admin)
                    self.outputs.append(admin)
                    admins.append(admin)

                else:
                    data = s.recv(1024)
                    if data:
                        if s in admins and self.sw in outputready:
                            self.sw.send(data)
                        elif s in outputready: # should be sw
                            self.process_sw_msg(data,s)
                    else:
                        s.close()
                        inputs.remove(s)
                        self.outputs.remove(s)
        self.serverS.close()
        self.serverN.close()

if __name__ == '__main__':
    portN = 4242
    portS = 6653
```

```
    try:
        opts, args = getopt.getopt(sys.argv[1:],"hn:s:",["nport=","
            sport="])
    except getopt.GetoptError:
        print 'srv_OF.py -n <northPort> -s <southPort>'
        sys.exit(2)
    for opt, arg in opts:
        if opt == '-h':
            print 'srv_OF.py -n <northPort> -s <southPort>'
            sys.exit()
        elif opt in ("-n", "--nport"):
            portN = int(arg)
        elif opt in ("-s", "--sport"):
            portS = int(arg)
    ServerOF(portN, portS).serve()
```

B Example d'administrateur client OpenFlow 1.3

```
#!/usr/bin/python
import getopt, socket, sys, time
from scapy.contrib.openflow3 import *

class ClientOF(object):

    def __init__(self, portN, host='127.0.0.1'):
        try:
            self.sock = socket.socket(socket.AF_INET, socket.
                SOCK_STREAM)
            self.sock.connect((host, portN))
            print 'Connected to server on port {0}'.format(portN)
        except socket.error, e:
            print 'Could not connect to server'
            sys.exit(1)

    def run_test(self):
        m = OFPTFeaturesRequest()
        self.sock.send(str(m))
        self.sock.close()

if __name__ == '__main__':
    portN = 4242
    try:
        opts, args = getopt.getopt(sys.argv[1:],"hn:",["nport="])
    except getopt.GetoptError:
        print 'srv_OF.py -n <northPort>'
        sys.exit(2)
    for opt, arg in opts:
        if opt == '-h':
            print 'srv_OF.py -n <northPort>'
            sys.exit()
        elif opt in ("-n", "--nport"):
            portN = int(arg)
    ClientOF(portN).run_test()
```