RFC 6056 : Transport Protocol Port Randomization Recommendations

Stéphane Bortzmeyer
<stephane+blog@bortzmeyer.org>

Première rédaction de cet article le 18 janvier 2011

Date de publication du RFC : Janvier 2011

https://www.bortzmeyer.org/6056.html

Lorsqu'un méchant attaquant tente de s'infiltrer dans une conversation en TCP, par exemple pour la couper ou la ralentir, il doit, soit être sur le trajet des paquets et pouvoir les regarder soit, dans le cas le plus fréquent, celui d'une attaque en aveugle, deviner un certain nombre d'informations qui « authentifient » la connexion TCP. Parmi celles-ci, le port source est trop souvent trop facile à deviner. Notre RFC expose pourquoi et surtout comment rendre ce port source plus dur à trouver.

De manière plus rigoureuse : lors d'une attaque en aveugle (où le méchant n'est pas en position de "*sniffer*" les paquets de la connexion TCP), les paquets mensongers injectés par le méchant ne seront acceptés par la machine victime que si le penta-tuple {Adresse IP source, Adresse IP destination, Protocole, Port Source, Post Destination} est correct (cf. RFC 5961 [1] et RFC 5927). Pour la sécurité de TCP face à ces attaques en aveugle, il vaut donc mieux que ce tuple soit caché. Mais une partie est triviale à deviner (par exemple, pour une connexion BGP, le port de destination sera forcément 179 dans un sens et, dans l'autre, ce sera le port source). Il faut donc faire porter l'effort sur les autres parties, en essayant de les rendre le moins prévisibles possible. Une telle protection fonctionnera aussi bien avec les attaques par TCP lui-même que en passant par ICMP <https://www.bortzmeyer.org/tcp-security.html>.

Bien sûr, cela ne remplace pas une bonne protection cryptographique des familles mais le moins qu'on puisse dire est qu'IPsec est laborieux à déployer et le but de notre RFC est de fournir une solution aujourd'hui. Il est à noter que les mécanismes de ce RFC peuvent être déployés unilatéralement et ne changent pas les protocole de transport (il n'est d'ailleurs pas spécifique à TCP ; ainsi, le DNS, protocole qui tourne surtout sur UDP, a dû déployer des techniques analogues, notamment en réponse à la faille

1. Pour voir le RFC de numéro NNN, https://www.ietf.org/rfc/rfcNNN.txt, par exemple https://www.ietf.org/rfc/rfc5961.txt

Kaminsky <https://www.bortzmeyer.org/comment-fonctionne-la-faille-kaminsky.html>; voir le RFC 5452).

La section 1 du RFC résume ce problème (voir aussi les RFC 4953, RFC 5927 et RFC 5961). Dans un protocole client-serveur, le protocole est souvent déterminé par l'application (BGP ne peut tourner que sur TCP), le port de destination est souvent fixe (80 pour HTTP), les adresses IP des serveurs en général connues. L'adresse IP du client n'est pas en général secrète (pour BGP, les adresses IP de tous les participants à un point d'échange sont un secret de Polichinelle). Il ne reste donc que le port source pour empêcher l'attaquant de fabriquer des faux paquets qui seraient acceptés. Certes, stocké sur 16 bits, il ne peut stocker, dans le meilleur des cas, que 65 536 valeurs différentes mais c'est mieux que rien. L'idée n'est pas de fournir une protection absolue mais simplement de rendre la tâche plus pénible pour l'attaquant.

Et, justement, il existe des moyens de faire varier ce port source de manière à rendre difficile sa prédiction par l'attaquant. Cela ne vaut pas des techniques comme l'authentification TCP du RFC 5925 mais c'est déjà un progrès pour la sécurité. En dépit du titre du RFC, il ne s'agit pas de rendre le port source réellement mathématiquement aléatoire mais simplement plus dur à deviner.

Place maintenant à l'exposé des méthodes. Juste un peu de vocabulaire : une « instance » est une communication identifiée par un penta-tuple (c'est équivalent à une connexion pour les protocoles de transport à connexions comme TCP). L'« ID » d'une instance est simplement le penta-tuple {Adresse IP source, Adresse IP destination, Protocole, Port Source, Post Destination}.

Il y a port et port. La section 2 explique le concept de « port temporaire ». Pour chaque protocole (TCP, UDP ou autre), les 65 536 valeurs possibles pour les ports se répartissent en « ports bien connus » (jusqu'à 1023 inclus), « ports enregistrés » (jusqu'à 49151 inclus) et « ports dynamiques et/ou privés » (le reste). Normalement, une application de type client/serveur se connecte à un « port bien connu » (parfois à un « port enregistré ») et utilise comme port source un port temporaire pris parmi les ports dynamiques. Pour que l'ID d'instance soit unique (sinon, il ne jouerait pas son rôle), une mise en œuvre simple est d'avoir une variable par protocole, mettons `next_ephemeral`, qui indique le prochain port libre et est incrémentée à chaque connexion. L'algorithme est en fait un peu plus compliqué que cela (section 2.2) car il faut tenir compte des ports déjà en service, et de ceux que l'administrateur système a manuellement interdit (par exemple, dans le serveur DNS Unbound, la plage de ports à utiliser se configure avec `outgoing-port-permit` et `outgoing-port-avoid`, dans son concurrent BIND, avec `use-v6-udp-ports` et `avoid-v6-udp-ports`; autre exemple, dans FreeBSD, la fonction `in_pcblookup_local()` vérifie les ports utilisés).

Mais le principal problème de cet algorithme simple est que le port source choisi pour la communication est facile à deviner. Si l'attaquant gère un serveur auquel le client se connecte, il lui suffit de regarder le port utilisé, d'ajouter un, et il saura le port utilisé pour la connexion suivante. Il pourra alors monter facilement les attaques décrites dans les RFC 4953 et RFC 5927.

Avant de trouver une solution à ce problème de prédictibilité, il faut se pencher sur le problème des collisions entre ID d'instances. Même si le client fait attention à utiliser un penta-tuple qui n'est pas en service **de son côté**, il peut parfaitement choisir un port source qui correspondra à un penta-tuple encore en activité de l'autre côté. En effet, l'autre côté doit garder la connexion (et donc le penta-tuple) en mémoire jusqu'à être sûr que plus aucun paquet IP n'est encore en route et, si le client est très actif (ouvre de nouvelles connexions souvent), le risque de tomber sur un penta-tuple encore en activité est élevé. Cela peut alors entraîner les problèmes décrits dans le RFC 1337. Choisir le prochain port de manière incrémentale, comme dans l'algorithme exposé plus haut, minimise le risque, puisqu'on ne réutilisera un port qu'après que toute la plage possible ait été épuisée.

Comment faire alors ? Faut-il choisir entre la sécurité qu'apportent une sélection non-linéaire du port source à utiliser et le sécurité par rapport aux collisions d'ID d'instances ? La section 3 expose plusieurs méthodes pour rendre le prochain port source plus difficile à prédire pour un attaquant. D'abord, qu'attend t-on d'un algorithme d'embrouillage du choix du port ? On voudrait à la fois qu'il rende la prédiction difficile pour l'attaquant, qu'il minimise le risque de collision d'ID, et qu'il ne marche pas sur les pieds des applications qui sont configurées pour un port précis (par exemple un serveur SSH qui écouterait sur le port 4222). Un algorithme qui assurerait la première fonction (rendre la prédiction difficile) pourrait, par exemple, être de tirer au hasard le prochain port utilisé. Mais les deux autres fonctions en souffriraient... N'imposons donc pas un vrai tirage aléatoire et voyons les compromis raisonnables. Le premier algorithme présenté, celui qui incrémente le numéro de port de un, a les problèmes inverses (il minimise la réutilisation mais maximise la prédictibilité).

On l'a vu, la plage des ports temporaires allait traditionnellement de 49152 à 65535. Mais elle est trop étroite pour assurer une vraie inprédictibilité du port source et la section 3.2 demande donc que, par défaut, la plage utilisée par les algorithmes de choix aille de 1024 à 65535. Comme cette plage inclus des ports réservés à l'IANA, l'algorithme de choix doit donc faire attention à ne pas sélectionner des ports qui sont utilisés sur la machine locale (ou qui pourraient l'être).

Place maintenant aux algorithmes effectifs. La section 3.3 décrit successivement les différents algorithmes possibles, avec du pseudo-code proche du C pour leur implémentation. L'algorithme 1, en section 3.3.1, est celui de simple tirage au hasard de la variable `next_ephemeral` à chaque utilisation. Si le port sélectionné n'est pas utilisable (par exemple parce qu'un serveur écoute déjà sur ce port), on essaie le suivant, puis le suivant etc. Attention, si on veut qu'il soit efficace, il faut que le générateur aléatoire le soit réellement (cf. RFC 4086). La traditionnelle fonction `random()`, dans la plupart des cas, ne l'est pas. Comme cet algorithme 1 n'a pas de mémoire, il peut sélectionner un port qui avait été utilisé récemment, augmentant ainsi les risques de collision des ID d'instances. En outre, il a une autre faiblesse : s'il existe une longue plage de ports déjà utilisés, et que le tirage au hasard fait tomber dans cette plage, le port situé immédiatement après la plage sera très souvent sélectionné, ce qui augmente la prévisibilité.

L'algorithme 2, en section 3.3.2, améliore l'algorithme 1 en tirant au hasard, non seulement le port initial, mais aussi l'incrément à utiliser si le port initial n'est pas libre. Il est donc le moins prévisible mais, en cas d'extrême malchance, il peut échouer à trouver un numéro de port, même s'il y en a un de libre.

L'algorithme 3 (section 3.3.3, inspiré du mécanisme décrit par Bellovin dans le RFC 6528) renonce à utiliser un générateur aléatoire et repose au contraire sur un condensat des paramètres de l'instance (adresses IP, etc). Prenons une fonction F qui reçoit les deux adresses IP, le numéro de port de l'autre partie et une clé secrète et les condensent en un nombre qui sera l'incrément appliqué à la variable `next_ephemeral`. En dépendant des paramètres de l'instance, on minimise la réutilisation des ID d'instances. On peut d'ailleurs ajouter d'autres paramètres comme, sur un système multi-utilisateurs, le nom de l'utilisateur. Cette fonction F doit évidemment être une fonction de hachage cryptographique comme MD5 (RFC 1321). Certes, MD5 a aujourd'hui des faiblesses cryptographiques connues mais il ne s'agit que de rendre le prochain port imprévisible, tâche pour laquelle il suffit. La clé secrète, elle, est ici pour empêcher un attaquant qui connaitrait la fonction de hachage utilisée de faire tourner le même algorithme et de prédire ainsi les numéros de port. Elle est typiquement choisie aléatoirement au démarrage de la machine.

L'algorithme 3 peut être perfectionné en ajoutant un second condensat, qui servira à indexer une table des variables `next_ephemeral` (au lieu d'avoir une variable unique). C'est l'algorithme 4, présenté en section 3.3.4.

Enfin, le dernier algorithme, le numéro 5 (section 3.3.5) choisit l'incrément de `next_ephemeral` au hasard mais l'applique ensuite de manière déterministe, ce qui limite les risques de collision d'ID

puisque le numéro de port utilisé augmente à chaque fois (et repart une fois arrivé au bout). Notez le rôle du paramètre N (dont le RFC dit qu'il devrait être configurable) pour déterminer si on utilise de petits incréments (plus prévisibles mais diminuant les risques de collision) ou des grands (avec les avantages et les inconvénients renversés).

Certains de ces algorithmes dépendent d'une clé secrète, choisie au hasard. Comment choisir celle-ci ? La section 3.4 qu'elle doit être assez longue (128 bits sont recommandés). Elle suggère aussi de changer cette clé de temps en temps. Mais attention : juste après le changement, le risque de collision va augmenter.

On l'a vu, il existe plusieurs algorithmes acceptables pour atteindre notre objectif de sélection de numéros de port peu prévisibles. Lequel choisir ? La section 3.5 est une évaluation comparée desdits algorithmes, basée sur le très bon article de Mark Allman, « "*Comments On Selecting Ephemeral Ports*" `<http://www.icir.org/mallman/papers/ports-ccr09.pdf>` », publié dans "*ACM Computer Communication Review*", 39(2), 2009. Tous ces algorithmes évitent les collisions de manière satisfaisante (moins de 0,3 % des cas mais le chiffre exact dépend de beaucoup de choses, comme le rythme d'ouverture et de fermeture des nouvelles connexions). L'étude empirique d'Allman montre que les algorithmes 1 et 2, les seuls réellement aléatoires sont aussi, comme attendu, ceux qui montrent le plus de collisions d'ID d'instances. Les algorithmes 3 et surtout le 4 sont ceux qui imposent la plus grande consommation de ressources machines (calcul du condensat cryptographique et, pour le 4, consommation de mémoire pour la table). À noter enfin que, dans certains systèmes d'exploitation et certaines applications (celles qui appellent `bind()` sans indiquer de numéro de port), les algorithmes 3 et 4 ne sont pas utilisables car l'adresse IP et le port distant ne sont pas connus au moment où on sélectionne le port local et la fonction de hachage ne dispose donc pas d'assez d'arguments. Le système d'exploitation doit alors se rabattre sur un algorithme comme le 2 ou bien effectuer un `bind()` paresseux en retardant l'allocation effective du port local jusqu'au moment où la connexion est réellement faite (avec, par exemple, `connect()`). C'est par exemple ce que fait Linux. (Pour l'anecdote, ce point avait été ma contribution à ce RFC, en 2008, au début du processus d'écriture.)

De nos jours, un grand nombre de machines sont coincées derrière un routeur NAT ou plus exactement **NAPT** ("*Network Address and Port Translation*", RFC 2663) qui utilise les numéros de ports pour démultiplexer le trafic entre les différentes machines du réseau local. Comme l'explique la section 4, les techniques d'obscurcissement du prochain numéro de port peuvent être alors prises en défaut si le routeur NAPT change ces ports par d'autres, choisis de manière trop prévisible. Le RFC impose donc que le routeur NAT, soit conserve les numéros de port (RFC 4787 et RFC 5382), soit choisisse lui-même les ports selon un mécanisme non prévisible.

Une fois les algorithmes exposés, compris et analysés, la section 5 reprend à nouveau de la hauteur pour parler d'une approche générale de la sécurité. D'abord, elle rappelle qu'un attaquant qui n'est pas aveugle, qui peut "*sniffer*" le réseau où passent les paquets, se moque des algorithmes exposés ici puisqu'il peut lire directement le numéro de port dans les paquets. Dans ce cas, il n'y a pas d'autre solution que des protections cryptographiques comme celles du RFC 4301.

Plus rigolo, certains choix pour la liste des ports exclus peuvent poser de sérieux problèmes de sécurité. Ainsi, si un système d'exploitation prend comme liste de ports à éviter la totalité de la liste IANA `<https://www.iana.org/assignments/port-numbers>`, il ne doit **pas** utiliser l'algorithme 1 car les ports qui se trouvent immédiatement après la liste réservée seront sur-représentés.

Comment est-ce mis en œuvre en pratique ? L'annexe A du RFC précise les choix effectués par les principaux systèmes d'exploitation libres (pour les autres, on ne sait pas s'ils sont sûrs puisqu'on ne peut pas examiner le source). Ainsi, Linux utilise l'algorithme 3, avec MD4 comme fonction de hachage (même si le RFC dit que c'est MD5). Sur un Linux 2.6 de septembre 2010, le code se trouve (de haut en bas, dans l'ordre des appels successifs), en indiquant fichier-source :fonction :

— net/ipv4/inet_hastables.c:inet_hash_connect (qui utilise net/ipv4/inet_connection_sock.c:inet_get_local_port_range pour récupèrer auprès de sysctl les ports utilisables),
— net/ipv4/inet_hastables.c:inet_sk_port_offset,
— drivers/char/random.c:secure_ipv4_port_ephemeral qui fait le hachage des paramètres indiqués par l'algorithme 3 (adresses et ports).

net/ipv4/inet_connection_sock.c:inet_csk_get_port, quant à elle, utilise l'algorithme 2 et non pas le 3. La clé secrète est dans un champ de la variable ip_keydata et elle est initialisée à une variable aléatoire par la fonction rekey_seq_generator.

FreeBSD, OpenBSD et OpenSolaris utilisent l'algorithme 1. La séquence des appels est (dans le HEAD du CVS de FreeBSD en octobre 2010) :

— in_pcbconnect,
— in_pcbconnect_setup,
— in_pcbbind_setup (qu'on peut voir, si on n'a pas les sources sur sa machine, en <http://fxr.watson.org/fxr/source/netinet/in_pcb.c?im=bigexcerpts#L325>).

Merci à Benoit Boissinot pour son aide dans la navigation dans le source de FreeBSD.

NetBSD, hélas, ne semble pas avoir de mécanisme pour rendre le prochaine numéro de port difficile à deviner (voir sys/netinet/in_pcb.c). Un "*patch*" avait été proposé <http://mail-index.netbsd.org/tech-net/2008/07/11/msg000624.html> mais apparemment jamais inclus. Cette remarque ne concerne que NetBSD utilisé sur une machine « terminale » car, sur un routeur NAT, NetBSD brouille bien (et sans intervention explicite de l'administrateur) les numéros de port.

Une petite anecdote historique pour terminer : la prédiction correcte des numéros de série des tanks allemands pendant la Seconde Guerre mondiale a permis aux Alliés de calculer le nombre de tanks ennemis <http://www.wired.com/autopia/2010/10/how-the-allies-used-math-against-german-tanks>. Comme quoi, toute information trop régulière et prévisible peut être exploitée par l'ennemi.