RFC 7873 : Domain Name System (DNS) Cookies

Stéphane Bortzmeyer
<stephane+blog@bortzmeyer.org>

Première rédaction de cet article le 10 juin 2016

Date de publication du RFC : Mai 2016

https://www.bortzmeyer.org/7873.html

La grande majorité des requêtes DNS passent aujourd'hui sur UDP. Ce protocole ne fournit aucun mécanisme permettant de vérifier un tant soit peu l'adresse IP source de la requête. Contrairement à ce qui arrive avec TCP, il est trivial de mentir sur l'adresse IP source, sans être détecté <https://www.bortzmeyer.org/usurpation-adresse-ip.html>. Cela permet des comportements négatifs, comme les attaques par réflexion <https://www.bortzmeyer.org/attaques-reflexion.html>. Ce nouveau RFC propose un mécanisme simple et léger pour s'assurer de l'adresse IP source du client : des petits gâteaux, les "*cookies*".

Le principe est le même que pour les "*cookies*" de HTTP (décrits dans le RFC 6265 [1]) : le serveur DNS génère un nombre imprévisible qu'il transmet au client. Celui-ci renvoie ce nombre à chaque requête, prouvant qu'il recevait bien les réponses, et n'avait donc pas triché sur son adresse IP. (Notez une grosse différence avec les "*cookies*" du Web : ils changent quand l'adresse IP du client change et ils ne peuvent donc pas être utilisés pour suivre à la trace un client mobile.)

Bien sûr, une autre solution serait d'utiliser TCP (comme proposé dans le RFC 7766) ou bien DTLS (RFC en cours de discussion). Mais les petits gâteaux se veulent une solution moins radicale, de déploiement peut-être plus facile. Ils ne sont pas forcément utilisés seuls, ils peuvent être combinés avec d'autres mesures anti-usurpation, comme celles du RFC 5452.

Comme avec toute technique de sécurité, il faut regarder en détail les menaces auxquelles elle répond (section 2 du RFC). En usurpant l'adresse IP source, un méchant peut effectuer des attaques par déni de service, et il peut **empoisonner un cache**. Voyons ces deux cas.

1. Pour voir le RFC de numéro NNN, https://www.ietf.org/rfc/rfcNNN.txt, par exemple https://www.ietf.org/rfc/rfc6265.txt

D'abord, l'attaque par déni de service : en usurpant l'adresse IP source, on peut effectuer une attaque par réflexion. Dans ces attaques, le méchant envoie un paquet à un tiers, le réflecteur, en mentant sur son adresse IP source : le méchant met celle de la victime. Lorsque le réflecteur répondra, il enverra un message à la victime. Cette attaque est surtout intéressante lorsqu'elle est combinée avec l'amplification <https://www.bortzmeyer.org/amplification-dns-combien.html>. Si la réponse est plus grosse que la question (ce qui est en général le cas avec le DNS), le méchant aura frappé la victime avec davantage d'octets que ce qu'il a lui-même envoyé.

Avec les "*cookies*", cette attaque ne serait pas possible, la réponse à une requête ayant un "*cookie*" erroné étant plus petite que la question.

Notez que les "*cookies*" ne protègent pas contre un attaquant situé sur le chemin ("*on-path attacker*") et qui peut lire le trafic réseau : voyant les paquets, il verra le "*cookie*" et pourra le transmettre. Les "*cookies*" n'empêchent donc pas toutes les attaques. D'autre part, si l'attaquant matraque directement le serveur DNS (sans réflexion), les "*cookies*" n'empêcheront pas l'attaque mais ils permettent de s'assurer que l'adresse IP source est exacte, ce qui autorisera de remonter à la source de l'attaque.

Le déni de service peut aussi viser le serveur DNS (au lieu de simplement l'utiliser comme réflecteur dans une attaque par réflexion). Chaque requête DNS va donner du travail au serveur et un nombre excessif peut dépasser ses capacités (comme dans l'attaque dafa888 <https://indico.dns-oarc.net/event/20/session/3/contribution/37>). Le problème est surtout aigu pour les serveurs récursifs, qui ont nettement plus à faire lorsqu'une requête arrive. Et ceux qui ont le plus de travail sont les serveurs récursifs qui valident avec DNSSEC : lors de la réception d'une réponse, il faut faire des calculs cryptographiques pour cette validation. Ces attaques par déni de service, au contraire de celles faites par réflexion, n'imposent pas de tricher sur l'adresse IP source mais, en le faisant, l'attaquant a l'avantage de rendre plus difficile son identification. Et cela peut lui permettre de faire traiter des requêtes qui seraient normalement refusées. Par exemple, si un résolveur n'accepte de requêtes que de son réseau (ce qui est la bonne pratique, cf. RFC 5358), et si on a oublié de filtrer en entrée les requêtes prétendant venir du réseau local, une attaque reste possible, en usurpant les adresses IP locales (une telle attaque est décrite dans l'exposé de Lars N[Caractère Unicode non montré [2]]ring <https://indico.dns-oarc.net/event/17/contribution/2/material/slides/0.pdf>).

Après les attaques par déni de service, voyons les attaques visant à faire accepter de fausses réponses, ce qui peut mener à empoisonner le cache. Le principe est de répondre à la place du vrai serveur faisant autorité, en usurpant son adresse IP. Si le méchant est plus rapide, sa réponse peut, dans certains cas, être acceptée par un résolveur (qui la mettra peut-être dans son cache, ce qui sera encore pire). L'attaque Kaminsky est une version améliorée de cette vieille attaque. Les "*cookies*" sont une bonne protection contre ce genre d'attaques.

Après les attaques, voyons les défenses. Il n'y a pas que les "*cookies*" dans la vie. D'abord, il y a DNSSEC (RFC 4034 et RFC 4035). DNSSEC permet d'authentifier les réponses DNS, résolvant ainsi complètement les attaques par empoisonnement. Par contre, il ne résout pas les attaques par déni de service et, pire, les calculs cryptographiques qu'il impose et la taille des réponses plus élevées peuvent dans certains cas aggraver une partie de ces attaques. (Le RFC ne le note pas, mais DNSSEC a une autre limite, que les "*cookies*" résolvent : s'il empêche l'empoisonnement, il ne permet pas pour autant d'obtenir la réponse DNS correcte. DNSSEC protège bien du hameçonnage, beaucoup moins de la censure <https://labs.ripe.net/Members/stephane_bortzmeyer/dns-censorship-dns-lies-seen-by-atlas-prob>.)

2. Car trop difficile à faire afficher par LaTeX

Autre solution de sécurité, TSIG (RFC 8945). TSIG est meilleur que les "*cookies*" dans la mesure où il permet de vérifier cryptographiquement l'identité de la machine avec qui on parle DNS. Mais il est non trivial à déployer : reposant sur de la cryptographie symétrique, il impose un partage des clés préalable. Cela le limite à des usages entre parties qui se connaissent bien (typiquement pour sécuriser les transferts de zone). On note aussi que, comme DNSSEC, mais contrairement aux "*cookies*", il nécessite des horloges synchronisées.

Pour résoudre ce problème de déployabilité, on peut envisager le mécanisme de distribution de clés TKEY (RFC 2930) ou bien passer à de la cryptographie asymétrique avec SIG(0) (RFC 2931). Mais aucune de ces deux techniques n'a connu de déploiement significatif.

Bref, les solutions de sécurité existantes ne résolvent pas réellement les problèmes que veulent traiter les "*cookies*". Mais assez parlé de la « concurrence », venons-en aux "*cookies*", comment marchent-ils (section 4 de notre RFC) ? Les "*cookies*" s'appuient sur EDNS (RFC 6891). Ils sont donc une option dans l'enregistrement EDNS. L'option "*cookie*" de EDNS porte le numéro 10 <https://www.iana.org/assignments/dns-parameters/dns-parameters.xml#dns-parameters-11>. Comme toutes les options EDNS, elle est codée en TLV : le type 10, la longueur et la valeur, qui comprend un ou deux "*cookies*". S'il n'y a que le "*cookie*" client, la longueur est fixe, de 8 octets. S'il y a en plus le "*cookie*" serveur, la longueur peut aller de 16 à 40 octets.

Le "*cookie*" client est normalement le résultat d'une fonction non-prévisible des adresses IP du client et du serveur, et d'un secret connu du client (par exemple, généré aléatoirement en suivant le RFC 4086, et changé de temps en temps). Cette fonction est, par exemple, une condensation mais le client prend ce qu'il veut : il est le seul à avoir besoin d'interpréter ses propres "*cookies*", ils sont opaques pour tout autre acteur. L'adresse IP du client est incluse dans les paramètres de la fonction notamment pour des raisons de vie privée : empêcher le client d'être reconnu s'il change d'adresse IP (le but de nos "*cookies*" DNS n'est pas du tout le même que celui des fameux "*cookies*" du Web).

Le "*cookie*" serveur prend comme paramètres de sa propre fonction (qui n'est pas forcément la même) l'adresse IP de son client, un secret (mêmes propriétés que chez le client), et le "*cookie*" du client (et pourquoi le "*cookie*" client ne se sert pas du "*cookie*" serveur ? Voyez la section 6.) Voilà comment on fabrique les gâteaux.

Mais comment les utilise-t-on ? La section 5 l'explique. Le client qui gère les gâteaux fabrique un "*cookie*" client qu'il envoie dans ses requêtes DNS. S'il n'a jamais parlé au serveur, il envoie une option "*cookie*" de forme courte, ne comprenant qu'un seul "*cookie*", le sien. S'il a déjà parlé au serveur et mémorisé le "*cookie*" de celui-ci, il fabrique une option EDNS "*cookie*" longue, incluant les deux "*cookies*".

Si le serveur ne comprend rien aux "*cookies*", il ne met pas l'option dans la réponse, et le client sait alors qu'il s'agit d'un vieux serveur, sans gestion des "*cookies*". (Ou bien c'était une attaque par repli ; le cas ne semble pas traité dans le RFC, la solution est sans doute que le client mémorise les serveurs "*cookie-capable*", pour détecter ces attaques.)

Si, par contre, le serveur gère les "*cookies*", il y a **cinq** possibilités :

— Si c'est le client qui est vieux, il n'envoie pas de "*cookie*". Le serveur répond alors comme aujourd'hui. Les "*cookies*" ne posent donc pas de problème d'interopérabilité : vieux et récents logiciels peuvent cohabiter.
— Si l'option EDNS "*cookie*" est présente, mais invalide (longueur inférieure à 8 octets, par exemple), le serveur répond `FORMERR` ("*FORmat ERRor*").

— Si la requête ne contient qu'un "*cookie*" client (client qui ne connaissait pas encore ce serveur), le serveur décide alors, selon sa politique à lui, de laisser tomber la requête (c'est violent, et cela implique de configurer le serveur avec les "*cookies*" des clients), d'envoyer un code d'erreur `BADCOOKIE` (valeur 23 <https://www.iana.org/assignments/dns-parameters/dns-parameters.xml#dns-parameters-6>), incluant le "*cookie*" du serveur, ou enfin de répondre normalement, en ajoutant son "*cookie*". En effet, dans ce cas, le client n'est pas « authentifié ». On n'a pas vérifié son adresse IP source. Il peut donc être justifié de ne pas donner la réponse tout de suite (le `BADCOOKIE`) ou bien, par exemple, de limiter le trafic de ce client (comme on le fait aujourd'hui <https://www.bortzmeyer.org/dns-rate-limiting-and-attacks.html>, avant les "*cookies*", puisqu'on n'est jamais sûr de l'adresse IP du client).
— Il y a deux "*cookies*" dans la requête mais le "*cookie*" serveur est incorrect, par exemple parce que le secret utilisé par le serveur a changé. C'est parfaitement normal et cela n'indique, ni une erreur, ni une attaque, simplement qu'on ne peut pas authentifier le client. On répond donc comme dans le cas précédent (avec trois choix, dont seuls les deux derniers sont réalistes).
— Il y a deux "*cookies*" dans la requête et le "*cookie*" serveur est correct. On a donc une certitude raisonnable que le client n'a pas usurpé son adresse IP (puisqu'il a reçu le bon "*cookie*" du serveur) et on va donc lui répondre. Les mesures de méfiance comme la limitation de trafic peuvent être débrayées pour cette requête.

Lorsqu'il reçoit une réponse, le client doit mémoriser le "*cookie*" du serveur. C'est particulièrement important la première fois, lorsque le client n'est pas encore authentifié. Si la réponse était `BADCOOKIE`, cela veut dire qu'on a affaire à un serveur grognon qui ne veut pas répondre sans qu'on lui donne un "*cookie*" correct : on retransmet alors la requête, cette fois en incluant le "*cookie*" transmis par le serveur.

Voilà, c'est tout. Avec ce système, on a une authentification légère et simple de l'adresse IP du client. Dans la vraie vie, il y aura peut-être quelques problèmes pratiques, que couvre la section 6 de notre RFC. Par exemple, si le client est derrière du NAT (RFC 3022), un méchant situé sur le même réseau local que lui pourrait faire une requête au serveur, obtenir le "*cookie*" du serveur et envoyer ensuite des requêtes en usurpant l'adresse IP locale du client légitime. Le serveur ne peut pas distinguer ces deux clients, le bon et le méchant. C'est pour cela que le "*cookie*" serveur inclut dans les paramètres de sa fonction le "*cookie*" du client. Ainsi, les deux machines, la gentille et la méchante auront des "*cookies*" serveur différents.

Un problème du même genre (plusieurs machines derrière une même adresse IP) pourrait survenir côté serveur, par exemple en raison de l'anycast. Mais on ne peut pas appliquer la même solution : si le "*cookie*" serveur dépend du "*cookie*" client et le "*cookie*" client du "*cookie*" serveur, on a une boucle sans fin. Le serveur doit donc se débrouiller : soit avoir le même secret (et donc les mêmes "*cookies*") sur toutes les machines (c'est l'approche la plus simple, et c'est celle recommandée par le RFC), soit faire en sorte qu'un client donné arrive toujours sur la même machine.

D'autres considérations pratiques figurent en section 7, notamment sur le remplacement d'un secret (ce qui invalidera les "*cookies*" précédemment distribués).

Et quelques discussions sur la sécurité, pour finir (section 9 du RFC). L'« authentification » fournie par les "*cookies*" est faible : elle ne protège pas contre un attaquant situé sur le chemin de communication entre client et serveur, lorsqu'il peut lire le trafic. Dans ce cas, l'attaquant a en effet accès au "*cookie*" et peut facilement le rejouer. Par exemple, si on est connecté à un réseau Wi-Fi public sans sécurité (pas de WPA), n'importe quel client du même réseau peut voir passer les "*cookies*". Néanmoins, les "*cookies*" réduisent quand même sérieusement l'ampleur du problème. Une attaque (l'usurpation d'adresse IP) que tout l'Internet pouvait faire est maintenant restreinte à un sous-ensemble de l'Internet. Si cela est encore trop, il faut passer à une sécurisation cryptographique comme celle que fournit le RFC 7858.

L'algorithme utilisé pour calculer les "*cookies*" est évidemment crucial. Il n'a pas besoin d'être normalisé, puisque seule la machine qui émet le "*cookie*" original a besoin de le comprendre. Mais il doit

garantir des "*cookies*" très difficiles à prévoir par un attaquant. On peut par exemple utiliser SHA-256 (RFC 6234, mais il n'y a pas forcément besoin de cryptographie top canon, les "*cookies*" n'étant qu'une authentification faible, de toute façon). Depuis la parution de notre RFC, le RFC 9018 a décrit un algorithme recommandé si des serveurs veulent être compatibles, par exemple s'ils font partie du même service "*anycast*".

Des exemples d'algorithmes figurent dans les annexes A et B (mais le RFC 9018 en a abandonné certains). Pour le client DNS, un algorithme simple serait d'appliquer la fonction simple et rapide FNV-64 à la concaténation des adresses IP du client et du serveur, et du secret. Un algorithme plus compliqué, mais plus sûr, serait de remplacer FNV par SHA-256, plus coûteux.

Pour le serveur, l'algorithme simple serait un FNV-64 de la concaténation de l'adresse IP du client, du "*cookie*" du client, et du secret. Pour l'algorithme compliqué, on peut tirer profit de la longueur plus grande du "*cookie*" serveur pour y mettre davantage d'informations : par exemple, huit octets calculés comme dans l'algorithme simple suivis de l'heure de génération du "*cookie*" (pour détecter plus facilement les vieux "*cookies*", avant même toute opération cryptographique).

Et les mises en œuvre ? Les "*cookies*" sont gérés par BIND à partir de la version 9.11 (pas encore officiellement publiée). Ils sont activés par défaut. Et Wireshark sait les afficher. Ici, un client nouveau (il ne connait pas encore le "*cookie*" serveur) :

```
No.     Time           Source                Destination           Protocol Length Info
      1 0.000000       192.168.2.9           192.168.2.7           DNS      97     Standard query 0x0000 SOA fool
...
Domain Name System (query)
...
    Additional records
        <Root>: type OPT
...
        Data length: 12
            Option: COOKIE
                Option Code: COOKIE (10)
                Option Length: 8
                Option Data: fb40ce9a68a6f1f0
                Client Cookie: fb40ce9a68a6f1f0
                Server Cookie: <MISSING>


No.     Time           Source                Destination           Protocol Length Info
      2 0.003910       192.168.2.7           192.168.2.9           DNS      200    Standard query response 0x000
...
      Additional records
        <Root>: type OPT
...
        Data length: 28
            Option: COOKIE
                Option Code: COOKIE (10)
                Option Length: 24
                Option Data: fb40ce9a68a6f1f0727e7501575acc5977ced0351ad20d56
                Client Cookie: fb40ce9a68a6f1f0
                Server Cookie: 727e7501575acc5977ced0351ad20d56
```

L'algorithme de condensation peut être choisi mais, apparemment, uniquement à la compilation (avec `--with-cc-alg=ALG` où `ALG` vaut `aes|sha1|sha256`).

Le dig livré avec cette version de BIND peut passer des "*cookies*" :

```
% dig +cookie @192.168.2.7 foobar.example
...
;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags: do; udp: 4096
; COOKIE: 51648e4b14ad9db8eb0d28c7575acbdde8f541a0cb52e2c2 (good)
;; QUESTION SECTION:
;foobar.example.                IN A
...
```

Pour les programmeurs en Go, la bibliothèque Go DNS <https://miek.nl/2014/August/16/go-dns-package/> gère désormais <https://github.com/miekg/dns/pull/350> les "*cookies*". Un exemple de code Go pour les envoyer :

```
m := new(dns.Msg)
m.Question = make([]dns.Question, 1)
c := new(dns.Client)
m.Question[0] = dns.Question{zone, dns.TypeSOA, dns.ClassINET}
o := new(dns.OPT)
o.Hdr.Name = "."
o.Hdr.Rrtype = dns.TypeOPT
o.Hdr.Class = 4096
e := new(dns.EDNS0_COOKIE)
e.Code = dns.EDNS0COOKIE
e.Cookie = "fb40ce9a68a6f1f0"
o.Option = append(o.Option, e)
m.Extra = make([]dns.RR, 1)
m.Extra[0] = o
```