RFC 9132 : Distributed Denial-of-Service Open Threat Signaling (DOTS) Signal Channel Specification

Stéphane Bortzmeyer
<stephane+blog@bortzmeyer.org>

Première rédaction de cet article le 10 octobre 2021

Date de publication du RFC : Septembre 2021

https://www.bortzmeyer.org/9132.html

Le protocole DOTS ("*Distributed Denial-of-Service Open Threat Signaling*") vise à permettre au client d'un service anti-dDoS de demander au service de mettre en route des mesures contre une attaque. Ce RFC décrit le canal de signalisation de DOTS, celui par lequel passera la demande d'atténuation de l'attaque. Il remplace le RFC 8782 [1], mais les changements sont mineurs.

Si vous voulez mieux comprendre DOTS, il est recommandé de lire le RFC 8612, qui décrit le cahier des charges de ce protocole, et le RFC 8811, qui décrit l'architecture générale. Ici, je vais résumer à l'extrême : un client DOTS, détectant qu'une attaque par déni de service est en cours contre lui, signale, par le canal normalisé dans ce RFC, à un serveur DOTS qu'il faudrait faire quelque chose. Le serveur DOTS est un service anti-dDoS qui va, par exemple, examiner le trafic, jeter ce qui appartient à l'attaque, et transmettre le reste à son client.

Ces attaques par déni de service sont une des plaies de l'Internet, et sont bien trop fréquentes aujourd'hui (cf. RFC 4987 ou RFC 4732 pour des exemples). Bien des réseaux n'ont pas les moyens de se défendre seuls et font donc appel à un service de protection (payant, en général, mais il existe aussi des services comme Deflect <https://deflect.ca/>). Ce service fera la guerre à leur place, recevant le trafic (via des manips DNS ou BGP), l'analysant, le filtrant et envoyant ce qui reste au client. Typiquement, le client DOTS sera chez le réseau attaqué, par exemple en tant que composant d'un IDS ou d'un pare-feu, et le serveur DOTS sera chez le service de protection. Notez donc que client et serveur DOTS sont chez deux organisations différentes, communiquant via le canal de signalisation ("*signal channel*"), qui fait l'objet de ce RFC.

1. Pour voir le RFC de numéro NNN, https://www.ietf.org/rfc/rfcNNN.txt, par exemple https://www.ietf.org/rfc/rfc8782.txt

La section 3 de notre RFC expose les grands principes du protocole utilisé sur ce canal de signalisation. Il repose sur CoAP, un équivalent léger de HTTP, ayant beaucoup de choses communes avec HTTP. Le choix d'un protocole différent de HTTP s'explique par les spécificités de DOTS : on l'utilise quand ça va mal, quand le réseau est attaqué, et il faut donc pouvoir continuer à fonctionner même quand de nombreux paquets sont perdus. CoAP a les caractéristiques utiles pour DOTS, il est conçu pour des réseaux où il y aura des pertes, il tourne sur UDP, il permet des messages avec ou sans accusé de réception, il utilise peu de ressources, il peut être sécurisé par DTLS...TCP est également utilisable mais UDP est préféré, pour éviter le "*head-of-line blocking*". CoAP est normalisé dans le RFC 7252. Parmi les choses à retenir, n'oubliez pas que l'encodage du chemin dans l'URI est un peu spécial, avec une option `Uri-Path:` par segment du chemin (RFC 7252, section 5.10.1). Par abus de langage, j'écrirai « le client CoAP demande `/foo/bar/truc.cbor` » alors qu'il y aura en fait trois options `Uri-Path:` :

```
Uri-Path: "foo"
Uri-Path: "bar"
Uri-Path: "truc.cbor"
```

Par défaut, DOTS va utiliser le port 4646 (et non pas le port par défaut de CoAP, 5684, pour éviter toute confusion avec d'autres services tournant sur CoAP). Ce port a été choisi pour une bonne raison, je vous laisse la chercher, la solution est à la fin de cet article. Le plan d'URI sera `coaps` ou `coaps+tcp` (RFC 7252, section 6, et RFC 8323, section 8.2).

Le fonctionnement de base est simple : le client DOTS se connecte au serveur, divers paramètres sont négociés. Des battements de cœur peuvent être utilisés (par le client ou par le serveur) pour garder la session ouverte et vérifier son bon fonctionnement. En cas d'attaque, le client va demander une action d'atténuation. Pendant que celle-ci est active, le serveur envoie de temps en temps des messages donnant des nouvelles. L'action se terminera, soit à l'expiration d'un délai défini au début, soit sur demande explicite du client. Le serveur est connu du client par configuration manuelle, ou bien par des techniques de découverte comme celles du RFC 8973.

Les messages sont encodés en CBOR (RFC 8949). Rappelez-vous que le modèle de données de CBOR est très proche de celui de JSON, et notre RFC spécifie donc les messages avec une syntaxe JSON, même si ce n'est pas l'encodage utilisé sur le câble. Pour une syntaxe formelle des messages, le RFC utilise YANG (cf. RFC 7951). Le type MIME des messages est `application/dots+cbor`.

La section 4 du RFC décrit les différents messages possibles plus en détail. Je ne vais pas tout reprendre ici, juste donner quelques exemples. Les URI commencent toujours par `/.well-known/dots` (`.well-known` est normalisé dans le RFC 8615, et `dots` est désormais enregistré à l'IANA <`https://www.iana.org/assignments/well-known-uris/well-known-uris.xml#well-known-uris-1`>). Les différentes actions ajouteront au chemin dans l'URI `/mitigate` pour les demandes d'actions d'atténuation, visant à protéger de l'attaque, `/hb` pour les battements de cœur, etc.

Voici par exemple une demande de protection, effectuée avec la méthode CoAP PUT :

```
Header: PUT (Code=0.03)
Uri-Path: ".well-known"
Uri-Path: "dots"
Uri-Path: "mitigate"
Uri-Path: "cuid=dz6pHjaADkaFTbjr0JGBpw"
Uri-Path: "mid=123"
Content-Format: "application/dots+cbor"

{
       ... Données en CBOR (représentées en JSON dans le RFC et dans
       cet article, pour la lisibilité).
}
```

L'URI, en notation traditionnelle, sera donc `/.well-known/dots/mitigate/cuid=dz6pHjaADkaFTbjr0JGBpw/m` CUID veut dire "*Client Unique IDentifier*" et sert à identifier le client DOTS, MID est "*Mitigation IDentifier*" et identifie une demande d'atténuation particulière. Si ce client DOTS fait une autre demande de palliation, le MID changera mais le CUID sera le même.

Que met-on dans le corps du message ? On a de nombreux champs définis pour indiquer ce qu'on veut protéger, et pour combien de temps. Par exemple, on pourrait avoir (je rappelle que c'est du CBOR, format binaire, en vrai) :

```
{
  "ietf-dots-signal-channel:mitigation-scope": {
    "scope": [
      {
        "target-prefix": [
           "2001:db8:6401::1/128",
           "2001:db8:6401::2/128"
         ],
        "target-port-range": [
          {
            "lower-port": 80
          },
          {
            "lower-port": 443
          }
         ],
         "target-protocol": [
           6
         ],
        "lifetime": 3600
      }
    ]
  }
}
```

Ici, le client demande qu'on protège `2001:db8:6401::1` et `2001:db8:6401::2` ("*target*" veut dire qu'ils sont la cible d'une attaque, pas qu'on veut les prendre pour cible), sur les ports 80 et 443, en TCP, pendant une heure. (`lower-port` seul, sans `upper-port` indique un port unique, pas un intervalle.)

Le serveur va alors répondre avec le code 2.01 (indiquant que la requête est acceptée et traitée) et des données :

```
{
   "ietf-dots-signal-channel:mitigation-scope": {
      "scope": [
         {
           "mid": 123,
           "lifetime": 3600
         }
       ]
    }
 }
```

La durée de l'action peut être plus petite que ce que le client a demandé, par exemple si le serveur n'accepte pas d'actions trop longues. Évidemment, si la requête n'est pas correcte, le serveur

répondra 4.00 (format invalide), si le client n'a pas payé, 4.03, s'il y a un conflit avec une autre requête, 4.09, etc. Le serveur peut donner des détails, et la liste des réponses possibles figure dans des registres IANA, comme celui de l'état d'une atténuation <https://www.iana.org/assignments/dots/dots.xml#dots-signal-channel-status-codes>, ou celui des conflits entre ce qui est demandé et d'autres actions en cours <https://www.iana.org/assignments/dots/dots.xml#dots-signal-channel-conflict-cause-codes>.

Le client DOTS peut ensuite récupérer des informations sur une action de palliation en cours, avec la méthode CoAP GET :

```
Header: GET (Code=0.01)
Uri-Path: ".well-known"
Uri-Path: "dots"
Uri-Path: "mitigate"
Uri-Path: "cuid=dz6pHjaADkaFTbjr0JGBpw"
Uri-Path: "mid=123"
```

Ce GET `/.well-known/dots/mitigate/cuid=dz6pHjaADkaFTbjr0JGBpw/mid=123` va renvoyer de l'information sur l'action d'identificateur (MID) 123 :

```
{
  "ietf-dots-signal-channel:mitigation-scope": {
    "scope": [
      {
        "mid": 123,
        "mitigation-start": "1507818393",
        "target-prefix": [
             "2001:db8:6401::1/128",
             "2001:db8:6401::2/128"
        ],
        "target-protocol": [
          6
        ],
        "lifetime": 1755,
        "status": "attack-stopped",
        "bytes-dropped": "0",
        "bps-dropped": "0",
        "pkts-dropped": "0",
        "pps-dropped": "0"
      }
    ]
  }
}
```

Les différents champs de la réponse sont assez évidents. Par exemple, `pkts-dropped` indique le nombre de paquets qui ont été jetés par le protecteur.

Pour mettre fin aux actions du système de protection, le client utilise évidemment la méthode CoAP DELETE :

```
Header: DELETE (Code=0.04)
Uri-Path: ".well-known"
Uri-Path: "dots"
Uri-Path: "mitigate"
Uri-Path: "cuid=dz6pHjaADkaFTbjr0JGBpw"
Uri-Path: "mid=123"
```

Le client DOTS peut se renseigner sur les capacités du serveur avec un GET de `/.well-known/dots/config`.

Ce RFC décrit le canal de signalisation de DOTS. Le RFC 8783, lui, décrit le canal de données. Le canal de signalisation est prévu pour faire passer des messages de petite taille, dans un environnement hostile (attaque en cours). Le canal de données est prévu pour des données de plus grande taille, dans un environnement où les mécanismes de transport normaux, comme HTTPS, sont utilisables. Typiquement, le client DOTS utilise le canal de données avant l'attaque, pour tout configurer, et le canal de signalisation pendant l'attaque, pour déclencher et arrêter l'atténuation.

Les messages possibles sont modélisés en YANG. YANG est normalisé dans le RFC 7950. Notez que YANG avait été initialement créé pour décrire les commandes envoyées par NETCONF (RFC 6241) ou RESTCONF (RFC 8040) mais ce n'est pas le cas ici : DOTS n'utilise ni NETCONF, ni RESTCONF mais son propre protocole basé sur CoAP. La section 5 du RFC contient tous les modules YANG utilisés.

La mise en correspondance des modules YANG avec l'encodage CBOR figure dans la section 6. (YANG permet une description abstraite d'un message mais ne dit pas, à lui tout seul, comment le représenter en bits sur le réseau.) Les clés CBOR sont toutes des entiers ; CBOR permet d'utiliser des chaînes de caractères comme clés mais DOTS cherche à gagner de la place. Ainsi, les tables de la section 6 nous apprennent que le champ `cuid` ("*Client Unique IDentifier*") a la clé 4, suivie d'une chaîne de caractères en CBOR. (Cette correspondance est désormais un registre IANA <https://www.iana.org/assignments/dots/dots.xml#dots-signal-channel-cbor-key-values>.) D'autre part, DOTS introduit une étiquette CBOR, 271 (enregistrée à l'IANA <https://www.iana.org/assignments/cbor-tags/cbor-tags.xml#tags>, cf. RFC 8949, section 3.4) pour marquer un document CBOR comme lié au protocole DOTS.

Évidemment, DOTS est critique en matière de sécurité. S'il ne fonctionne pas, on ne pourra pas réclamer une action de la part du service de protection. Et s'il est mal authentifié, on risque de voir le méchant envoyer de faux messages DOTS, par exemple en demandant l'arrêt de l'atténuation. La section 8 du RFC rappelle donc l'importance de sécuriser DOTS par TLS ou plutôt, la plupart du temps, par son équivalent pour UDP, DTLS (RFC 6347). Le RFC insiste sur l'authentification mutuelle du serveur et du client <https://twitter.com/LlanOldDog/status/661413291126714368>, chacun doit s'assurer de l'identité de l'autre, par les méthodes TLS habituelles (typiquement via un certificat). Le profil de DTLS recommandé (TLS est riche en options et il faut spécifier lesquelles sont nécessaires et lesquelles sont déconseillées) est en section 7. Par exemple, le chiffrement intègre est nécessaire.

La section 11 revient sur les questions de sécurité en ajoutant d'autres avertissements. Par exemple, TLS ne protège pas contre certaines attaques par déni de service, comme un paquet TCP RST ("*ReSeT*"). On peut sécuriser la communication avec TCP-AO (RFC 5925) mais c'est un vœu pieux, il est très peu déployé à l'heure actuelle. Ah, et puis si les ressources à protéger sont identifiées par un nom de domaine, et pas une adresse ou un préfixe IP (`target-fqdn` au lieu de `target-prefix`), le RFC dit qu'évidemment la résolution doit être faite avec DNSSEC.

Question mises en œuvre, DOTS dispose d'au moins quatre implémentations, dont l'interopérabilité a été testée plusieurs fois lors de hackathons IETF (la première fois ayant été à Singapour, lors de l'IETF 100 <https://www.ietf.org/how/meetings/100/>) :

— Le travail <https://github.com/nttdots/go-dots> de NTT, en Go, en logiciel libre,
— Les travaux non-libres de NCG <https://www.nccgroup.trust/>, Arbor et Huawei

Notez qu'il existe des serveurs de test DOTS publics <https://trac.ietf.org/trac/dots/> comme `coaps://dotsserver.ddos-secure.net:4646`.

Ah, et la raison du choix du port 4646 ? C'est parce que 46 est le code ASCII pour le point ("*dot*" en anglais) donc deux 46 font deux points donc "*dots*".

L'annexe A de notre RFC résume les principaux changements depuis le RFC 8782. Le principal changement touche les modules YANG, mis à jour pour réparer une erreur et pour tenir compte du RFC 8791. Il y a aussi une nouvelle section (la 9), qui détaille les codes d'erreur à renvoyer, et l'espace des valeurs des attributs a été réorganisé. . .Rien de bien crucial, donc.