Thibaut Passilly
thibaut.passilly@epita.fr
EPITA SRS 2019
Cryptologie
15 mai 2018

Génération de TOTP

avec Google Authenticator

Abstract

Google Authenticator lets users open sessions of many applications such as SSH, Facebook or even OVH, with a one-time password, a random number of 6 digits that doesn't require any network synchronization. This system is called TOTP and it relies on three technical principles : first generation of the secret key according to a random event and sharing between the client and the server, computation of the numeric signature based on this key and on HMAC-SHA1, then the time synchronization between the application and the server that permits to decode the token. However, while this is considered a secure procedure by its producers, we may discuss the cryptographic level of the solution. The use of SHA1 is a little bit conflicting with the Google's past. As a secure precaution, it is very recommended to use TOTP rather than HOTP, and only if it is part of a multi-factor authentication process. The complexity of brute force attacks on the six-digits token is discussable, and some opponents to Google's application tend to propose some other solutions such as FIDO/U2F.

Sommaire

1. Présentation des TOTP et principes de base

En 2016, la première version de l'application Google Authenticator voit le jour. Cette application permet à un utilisateur de se connecter à un service en utilisant un mot de passe unique. Parmi ces services, on trouve le fameux protocole de *shell* sécurisé à distance, SSH, les sessions de plusieurs applications Facebook, et encore beaucoup d'autres exemples disponibles notamment sur la page Wikipédia [1] de Google Authenticator. Ce dernier fait partie des générateurs de TOTP. Ceci se traduit par *Time-Based One-Time Password*, autrement dit « mot de passe à utilisation unique basé sur le temps ». Les normes quant aux TOTP sont définies dans la RFC 6238 [2] de l'IETF. Cette référence nous mentionne d'ailleurs une information importante en considération de la valeur cryptographique des TOTP. Un mot de passe d'un tel type est une extension de la technologie OTP, *One-Time Password*, et vient compléter le principe très important de HOTP, *HMAC-Based One-Time Password*, un autre descendant des OTP. Pour parler de TOTP, il faut donc au premier plan évoquer les HOTP. Ces types de mots de passe sont eux-mêmes définis dans la RFC 4226 [3].

1. a. HOTP

Parmi les points importants abordés par la RFC 4226, on retrouve à la fois l'algorithme utilisé et ses pré-requis, des considérations au niveau de la sécurité de la solution, et surtout, dans la description de l'algorithme, le fait que HOTP est un algorithme à clés symétriques. Comme le dit la RFC, *the ability to embed the algorithm into high-volume SIM and Java cards was a fundamental prerequisite*. En d'autres termes, cette fonctionnalité est prévue entre autres pour des systèmes embarqués tels que les téléphones, ce qui nécessite une adaptation au niveau des capacités matérielles notamment.

De plus, un HOTP doit être « d'une longueur minimale de 6 chiffres ». La condition selon laquelle ce *token* doit être uniquement composé de chiffres est une option. On peut donc remplacer « chiffres » par octets. Quant au secret partagé, constituant la symétrie de l'algorithme, la RFC 4226 exige une clé d'au moins 128 bits, en recommandant toutefois une longueur de 160 bits. Au niveau de l'algorithme, celui-ci se résume au calcul suivant :

```
HOTP(K,C) = Truncate(HMAC-SHA-1(K,C))
```

Figure 1 : Calcul du HOTP

Ici, `K` représente le secret partagé par le client et le serveur et `C` représente le compteur, qui doit absolument être synchronisé entre les deux entités. Cette variable sert à recenser le nombre de requêtes de la part de l'utilisateur afin de générer un nouveau mot de passe. Cette métrique est importante, car elle définit la manière de synchroniser le client et le serveur.

Pour résumer le principe de synchronisation pour HOTP, on peut considérer le principe suivant : le serveur conserve une valeur maximale correspondant à un écart, que l'on appellera `w`. Cet écart représente le nombre maximal de générations que le client a effectué lorsque le client et le serveur n'étaient pas synchronisés. Pour se resynchroniser, le serveur itère sur les valeurs allant de `C` à `C + w`, en comparant le résultat du calcul du HOTP avec la valeur courante du client. `Truncate` représente une fonction permettant d'extraire quatre octets de l'élément passé en paramètre, d'une manière prédéfinie. Enfin, `HMAC-SHA-1` est la fonction qui nous intéresse. Il s'agit plus précisément de la fonction *Hash-Based Message Authentication Code*, c'est-à-dire un mécanisme pour l'authentification de messages utilisant les fonctions de hachage cryptographique.

1. b. HMAC

HMAC est définie dans la RFC 2104 [4]. Celle-ci correspond en fait à l'opération suivante :

```
HMAC(K,text) = H(K' xor opad, H(K' xor ipad, text))
```

Figure 2 : Calcul du HMAC

Ici, `H` représente une fonction de hachage. Pour HOTP, `H = SHA1`. `opad` (*outer pad*) et `ipad` (*inner pad*) représentent respectivement les octets 0x36 et 0x5c répétés chacun `B` fois, `B` étant la taille (en octets) des blocs sur lesquels itère la fonction `H`. Pour SHA1, on aura donc `B = 512 / 8 = 64`. `K` représente le secret partagé par le client et le serveur et `K'` est en fait le dérivé de `K`, dans le sens où celui-ci peut correspondre à :

- `K` *paddé* par la droite avec des `0` pour avoir la taille de `B`, dans le cas où `K` a une taille inférieure à celle de `B`;
- `K` haché grâce à la fonction de hachage `H`, ce qui permet d'atteindre la taille de `B`.

Toutefois, le premier cas est à proscrire, car selon la RFC 2104, *less than L bytes is strongly discouraged as it would decrease the security strength of the function*, `L` correspondant à la taille (en octets) de sortie de la fonction de hachage utilisée; en l'occurence, `L = 20`. De même, *keys longer than L bytes are acceptable but the extra length would not significantly increase the function strength*. Pour résumer, utiliser une clé d'une taille supérieure à celle de la sortie de la fonction de hachage utilisée (dans notre cas, 20) semble être une bonne idée.

On peut se demander pourquoi les valeurs 0x36 et 0x5c sont utilisées pour, respectivement, `opad` et `ipad`. Il s'agit en fait de *magic numbers* et ceux-ci traduisent en quelques sortes la sécurité de HMAC. Dans la RFC 2104 datant de 1997, M. Bellare, R. Canetti, et H. Krawczyk ont expliqué que ces chiffres ont en fait été choisis arbitrairement, et permettent d'avoir des résultats différents pour `K' xor opad` et `K' xor ipad`. De cette manière, deux autres chiffres, à conditions qu'ils ne soient pas identiques, auraient pu être choisis.

1. c. SHA1

Évoquons désormais la notion de fonction de hachage. Celle-ci se définit par une fonction qui associe une donnée entrante (typiquement, des octets) à une sortie de taille fixe. Nous allons voir plus précisément le cas de SHA1, tel que défini dans la RFC 3174 [5].
Celui-ci se décrit en trois étapes :

1. L'entrée de cet algorithme est un entier inférieur à 2^{64}, donc une taille maximale de 64 bits.

2. L'algorithme calcule ensuite un *payload* de la manière suivante :

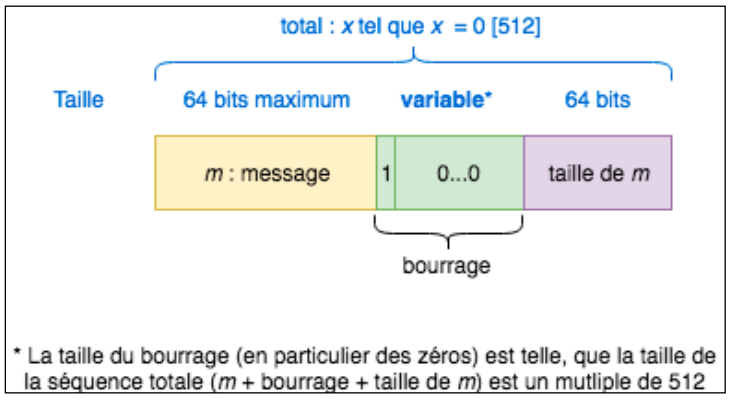

Figure 3 : Schéma de la charge utile pour SHA1

Le bourrage est très utile, car il permet ensuite à l'algorithme de travailler sur des blocs de taille fixe (512 bits, bien entendu), appelés M_X.

3. On définit cinq *buffers* H_X, X compris entre 0 et 4, en hexadécimal :

- H_0 = 67452301
- H_1 = EFCDAB89
- H_2 = 98BADCFE
- H_3 = 10325476
- H_4 = C3D2E1F0

Que signifient ces valeurs ? Ce sont en fait des nombres considérés comme *Nothing up my sleeve*, c'est-à-dire qu'ils sont censés ne pas avoir de propriétés spéciales, afin notamment d'éviter le *backdooring*, qui pourraient permettre par exemple à un organisme étatique d'être capable de déchiffrer des condensats SHA1.

Le principe de l'algorithme est d'effectuer, pour chaque bloc M_i de 512 bits, les procédures suivantes, 80 fois en tout (selon la méthode 1 de la RFC en question):

3. a. Décomposer M_i en 16 mots (4 octets chacun), appelés W_j, avec W_0 le mot le plus à gauche, c'est-à-dire les 32 bits de poids fort de M.

3. b. Ensuite, on peut déployer l'algorithme suivant (en pseudo-code):

```
Pour t allant de 16 à 79
      Wt = clshift(1,Wt-3 XOR Wt-8 XOR Wt-14 XOR Wt-16)
A, B, C, D, E = H0, H1, H2, H3, H4
Pour t allant de 0 à 79
      TEMP = clshift(5, A) + f(t;B,C,D) + E + Wt + Kt
      E,D,C,B,A = D,C,clshift(30, B),TEMP
H0,H1,H2,H3,H4 = H0 + A,H1+B,H2+C,H3+D,H3+E
```

Figure 4 : Algorithme SHA1

Ici, `clshift` est une fonction permettant d'effectuer une rotation circulaire par la gauche. Celle-ci prend deux paramètres, un entier représentant le nombre de *shift* en question, et le deuxième étant le message à *shifter*.

K_t correspond en fait à 4 mots, définis tels que :

- K_t = 5A827999, 0 <= t <= 19
- K_t = 6ED9EBA1, 20 <= t <= 39
- K_t = 8F1BBCDC, 40 <= t <= 59
- K_t = CA62C1D6, 60 <= t <= 79

Enfin, `f(t;B,C,D)` est une fonction qui vaut :

- (B AND C) OR ((NOT B) AND D), pour 0 <= t <= 19
- B XOR C XOR D, pour 20 <= t <= 39
- (B AND C) OR (B AND D) OR (C AND D), pour 40 <= t <= 59
- B XOR C XOR D, pour 60 <= t <= 79

Ces étapes sont répétées 80 fois pour une version « classique » de SHA1; il existe toutefois des variantes à 53 tours.

Finalement, le message obtenu est la concaténation des H_X (avec le X le plus petit à gauche), et fait donc une taille constante de 160 bits. À noter que les différentes méthodes (dont deux définies explicitement dans la RFC) donnent, bien entendu, le même résultat.

1. d. TOTP

L'étude des précédents éléments nous amène aux TOTP, comme décrits dans la RFC 6238. Les mots de passes uniques prenant en compte le temps sont une extension des HOTP, et plus précisément une variante de l'algorithme utilisé pour les HOTP. Pour décrire l'algorithme TOTP, il faut donc se baser sur celui du HOTP (voir *Figure 1 : Calcul du HOTP*), notamment en modifiant le paramètre `C` qui représente le compteur, c'est-à-dire le nombre de fois où le client a demandé un nouveau mot de passe. Ici, `C` sera une variable qui dépend du temps. On l'appellera `T` pour plus de clarté.

On aura donc :

- X le pas en secondes; par défaut, celui-ci vaut 30. C'est un paramètre du système d'exploitation.

- T est un entier qui représente le nombre de pas écoulés entre le moment T_0 et la date actuelle. Par exemple, si le pas vaut 30 secondes et que le temps écoulé vaut 1:20 min, alors le nombre de pas écoulés est 2. À 1:30 min, il vaudra donc 3.

À noter que T_0 vaut par défaut 0, ce que l'on appelle *Unix epoch*, ou plutôt la date initiale à partir de laquelle est mesuré le temps dans les systèmes d'exploitation, soit le 1er janvier 1970 à 0h00 (UTC). Il existe différentes *epoch*, mais le TOTP se basera sur celui d'Unix.

Enfin, évoquons la synchronisation entre le client et le serveur. Outre les problèmes de temps-réel qui seront évoqués dans la **partie 3**, plusieurs paramètres sont à prendre en compte pour la synchronisation. À l'instar du HOTP, le compteur (ici T) sera le paramètre principal de synchronisation entre le client et le serveur, et fonctionnera donc de la même manière que pour le HOTP : un délai maximal est initialisé sur le serveur, qui retrouve la valeur du TOTP en itérant sur les différentes valeurs de T.

1. e. Génération de la clé secrète

En théorie, le premier secret (C et T pour, respectivement, HMAC et TOTP) doit provenir d'un événement aléatoire, sinon la suite de nombres générés depuis ce secret ne seraient pas aléatoires à proprement parler. La génération de cette première clé peut dépendre d'événements aléatoires de deux types, comme décrits dans la RFC 4226 :

- logiciel : une source logicielle est utilisée pour générer des événements aléatoires. On peut penser à des valeurs imprévisibles générées par les différentes applications. Typiquement, la génération des nombres aléatoires basée sur les *race conditions* et plus particulièrement l'imprédictabilité des changements de contexte, est un exemple pertinent de source logicielle de génération de secrets. Un papier [9] sur le sujet a été réalisé par une équipe de chercheurs de l'Université de Cluj-Napoca, en Roumanie, apportant de nombreuses preuves illustrées par des tests, ayant obtenu par ailleurs une note de « 90 % aux tests de NIST ». Nous reviendrons sur ces tests dans la partie suivante.

- matériel : ce thème exploite l'aléatoire causé par des phénomènes physiques. Les plus souvent cités sont la température d'un ordinateur ou d'un périphérique, la vitesse de rotation d'un disque dur, etc.

De plus, l'utilisation d'un algorithme de génération de nombres aléatoires voit toute son utilité dans la diversité des nombres qu'il produit. En effet, un individu malveillant pourrait s'amuser à faire varier volontairement un paramètre logiciel ou matériel jugé aléatoire pour être capable de prévoir les futurs événements. Par exemple, en augmentant volontairement la température d'un ordinateur, et en partant du fait que le secret est généré à partir de la température mesurée par l'ordinateur, il est possible pour l'individu de prévoir les nombres aléatoires produits. En revanche, l'utilisation d'un générateur de nombres aléatoires « par-dessus » apporte une couche d'entropie conséquente et donc indispensable.

1. f. Tester l'aléatoire

Pour tester l'efficacité et la crédibilité de telles méthodes, plusieurs tests connus ont été réalisés. Parmi les plus connus, les tests de Diehard (G. Marsaglia) ou encore les tests de NIST, évoqués précédemment, procurent des résultats convaincants.

Pour Diehard, les tests sont regroupés en plusieurs thèmes, notamment :

- l'espace ou la distance entre les valeurs générées
- le taux de répétition de motifs
- le taux de présence / absence de motifs

Enfin, concernant les tests de NIST, aussi appelés « NIST SP 800-22 », ceux-ci vont permettre de vérifier, entre autres, les caractéristiques suivantes :

- la fréquence de bits et de motifs
- l'entropie des différents résultats
- la taille de la plus grande suite d'un même bit

La test suite NIST[1] est disponible gratuitement à l'adresse en bas de page.

Les principes énoncés ci-avant sont les principes théoriques de TOTP et HOTP, et de la notion d'aléatoire. Intéressons-nous désormais au cas particulier de Google Authenticator.

[1] https://csrc.nist.gov/projects/random-bit-generation/documentation-and-software

2. Fonctionnement de Google Authenticator

Dans cette partie, nous allons voir différents aspects du fonctionnement de Google Authenticator, en détaillant chaque étape, depuis la création du secret partagé, jusqu'à la génération de *tokens*.

Pour commencer, il est indispensable de connaître le fonctionnement général de l'outil. Google Authenticator supporte à la fois HOTP et TOTP, en suivant les RFC 4226 et 6238 (information testée et issue du GitHub officiel [6] du projet). En effet, lors de la création d'une nouvelle session, le programme nous demande si on veut que le *token* soit basé sur le temps. La première étape consiste donc à générer un code alphanumérique. À partir de ce code, le client et le serveur vont tous deux générer les mêmes OTP. La synchronisation n'est pas nécessaire, nous allons voir pourquoi.

2. a. Génération du secret partagé

La génération du code alphanumérique est en fait l'étape pendant laquelle est créé le secret partagé entre le client et le serveur. Cette étape, contrairement aux suivantes, ne se fait pas directement dans l'application pour *smartphone*. Pour cette première partie, la génération du secret pour le client et le serveur se fera selon la RFC 6238, comme vu précédemment. Pour résumer, on énumère les étapes :

- choix d'un paramètre secret
- calcul du HMAC à partir de celui-ci
- calcul du HOTP

Nous allons voir ici plus précisément la phase antérieure à toutes ces étapes, c'est-à-dire la phase de génération de la clé secrète, évoquée dans la partie 7.5 de la RFC 4226. Deux options sont possibles pour la génération de cette clé :

- génération déterministe : les secrets dérivent d'une « master key ».
- génération aléatoire : les secrets sont générés aléatoirement et sont stockés immédiatement.

Pour Google Authenticator, on sera dans le premier cas : la clé « master » est générée, puis stockée, et les suivantes (les codes aléatoires) sont générées depuis cette « master key ».

Comme vu dans les parties précédentes, le secret partagé est au départ issu d'une génération purement aléatoire (cf. 1.e).

Une fois que ce secret est généré, il doit être désormais partagé. Ce mécanisme de partage peut se faire de deux manières :

- en scannant un QR Code
- en visitant une URL

Dans les deux cas, c'est bien entendu la même information qui est présente.

À noter que le secret est encodé en base32 (RFC 4648). Cela est dû au fait que tous les caractères ne sont pas forcément représentables, ce qui poserait problème, en particulier pour la génération de l'URL et l'accès à celle-ci. On peut alors se poser la question : pourquoi ne pas avoir utilisé de base64 ? Même si les développeurs restent assez silencieux sur ce sujet, on peut toutefois noter que la base32 est souvent préférée à la base64 pour trois raisons principales :

- tous les caractères possibles tiennent en une seule casse (pas de minuscules)
- le caractère '/' n'est pas encodé, ce qui limite les problèmes pour les noms de fichiers, et plus particulièrement, dans notre cas, des URL
- la RFC 4648 tient compte des ressemblances visibles entre les caractères et omet volontairement les chiffres '1', '8' et '0' car ils ressemblent trop aux lettres 'I', 'B', et 'O'

- en excluant le padding (caractère '=' dans la RFC en question), on peut encoder facilement des URL sans rien encoder par-dessus.

On pourrait alors évoquer la base16, cependant elle ne procure pas plus d'avantages pertinents et prend environ 20% de mémoire en plus.

2. b. Synchronisation unique client-serveur

Cette section a été appelée « synchronisation unique » car l'étape de synchronisation ne se fait qu'une fois entre le client et le serveur et ne se refera plus par la suite. Cela signifie que s'il arrive un problème quelconque à l'une des deux entités en ce qui concerne sa gestion du temps, la synchronisation sera obsolète et non re-synchronisée. Toutefois, le serveur peut s'adapter au client, comme énoncé dans la fin de la partie 1.d : le serveur dispose d'un intervalle d'acceptation du retard, et va alors comparer tous les tokens générés dans un certain intervalle de temps jusqu'à retrouver le token actuel. Cependant, pour des raisons de sécurité, cet intervalle de temps est très faible et ne permet absolument pas de gérer un réel dysfonctionnement du temps dans l'une des deux machines.

Ceci fait à la fois la force et la faiblesse des TOTP comme Google Authenticator, nous verrons dans la partie **Cryptanalyse** plus en détails cet aspect.

2. c. Signature

Une fois la synchronisation unique effectuée, l'outil est opérationnel et peut désormais générer des mots de passes uniques prenant en compte la métrique temporelle. Pour ce faire, Google Authenticator utilise HMAC-SHA1, comme décrit dans les parties 1.b et 1.c.

2. d. Algorithme de décodage

En guise de résumé du fonctionnement de Google Authenticator, nous pouvons considérer l'algorithme suivant pour le décodage en mode TOTP :

```
google_authenticator_decode(shared):
      secret = b32decode(upper(rm_spaces(shared)))
      input = current_unix_time() / 30
      hmac = sha1(secret + sha1(secret + input))
      bytes = hmac[last_byte(hmac):last_byte(hmac) + 4]
      token = int(bytes) mod power(10,6)
```

Figure 5 : Pseudo-algorithme de décodage du token en TOTP (issu du site garbagecollected.com [7])

Dans cet algorithme, `shared` correspond au secret partagé, une clé de 32 bits. Ensuite, cette suite d'instructions commence par le décodage de ce secret. On rappelle ici que le secret est en base32 pour les raisons énoncées à la fin de la partie 2.a. La particularité de Google Authenticator est de sauvegarder le secret avec des espaces et en minuscules, afin qu'il soit plus lisible pour les humains; ceci est pratique au moment de la génération de ce secret et donc de la synchronisation avec le serveur (cf. 2.b).

Une fois que le secret est décodé, on a besoin de la métrique temporelle. Pour cela, on suit la RFC 6238 (cf. 1.d) en utilisant le *Unix epoch*. On remarque alors que le résultat est divisé par 30. Cela fait écho à l'intervalle de 30 secondes nécessaire au serveur pour se synchroniser en testant toutes les entrées des 30 dernières secondes. À noter que cela est possible car l'*epoch* est en fait stocké en secondes. L'entrée correspondra donc au temps.

La partie la plus intéressante de cet algorithme réside dans les trois dernières lignes. Le condensât HMAC est calculé, à l'aide de l'algorithme de hachage SHA1. La formule utilisée dans l'algorithme a volontairement été simplifiée. Ceci étant, un TOTP, comme le spécifie la RFC 6238, doit être composé d'au moins six chiffres. Google Authenticator a décidé de prendre le « minimum légal » en choisissant six.

Ainsi, on ne va pas directement prendre les 20 octets produits par HMAC-SHA1, ce qui correspond à 40 caractères hexadécimaux. Nous allons donc sélectionner 4 octets de notre condensât, ce qui nous donnera 32 bits, soit un entier non signé d'une valeur maximale de $2^{32} - 1$.

En guise d'indice pour faire notre choix des 4 octets, celui-ci sera également déterminé pseudo-aléatoirement vu que l'indice est calculé par rapport au dernier mot du résultat du HMAC. Cela nous procure un entier sur 4 bits, soit une valeur strictement inférieure à 16, ce qui nous permet de ne jamais déborder sur la taille du HMAC, car celui-ci a une taille de 20 octets (il est issu de SHA1); même si les 4 derniers bits sont tous à 1, on prendra donc les octets d'indices 15 à 19, ce qui ne déborde pas, au maximum, sur la taille du HMAC. Une fois l'extrait de HMAC obtenu, il ne reste plus qu'à en faire un entier 32 bits non signé, auquel nous appliquerons un modulo 10^6 afin d'obtenir un entier décimal de 6 chiffres constituant notre OTP.

Une implémentation en Python3 est disponible en supplément, réalisée par mes soins. Je me suis servi des bibliothèques *hmac*, *time*, *hashlib*, et *base64*.

De plus, une version de l'encodage en base32 est disponible en annexes de ce rapport, issue du GitHub officiel de Google Authenticator, pour la version Java (Android) de l'application.

3. Cryptanalyse et problèmes d'implémentation

Plusieurs des aspects présentés sont critiquables au niveau cryptographique.

3. a. HMAC-SHA1

De façon évidente, il est possible de critiquer SHA1. En effet, cette fonction de hachage est connue pour ses torts.

En 2005, soit 5 ans après son invention, Vincent Rijmen et Élisabeth Oswald, deux cryptologues de renom, publient une attaque sur SHA1 dans sa version 53 tours, permettant de trouver une collision en moins de 2^{80} opérations. La même année, une équipe de cryptologues chinois, comptant parmi eux la célèbre Wang Xiaoyun, publie une attaque sur la version « complète », c'est-à-dire celle de 80 tours, permettant de trouver une collision en 2^{69} opérations. Cette équipe a, par la même occasion, publié une attaque sur la version « simplifiée » de SHA1 (53 tours), permettant de trouver une collision en 2^{33} tours. En août, toujours la même année, Wang Xiaoyun annonce une amélioration de son attaque, passant de 2^{66} à 2^{63} opérations.

Ces événements montrent sans aucun doute la faiblesse d'un tel algorithme, mais le fait le plus marquant et peu anodin est sans doute le suivant : l'année dernière (janvier 2017), une équipe composée de chercheurs de Google et du CWI (Centrum Wiskunde et Informatica), un centre de recherches hollandais en mathématiques et en informatique théorique, découvrent une collision cette fois-ci non pas via des calculs de complexité, mais entre deux documents PDF. Ces deux documents, pourtant différents, présentent le même hachage via SHA1.

La question que l'on peut se poser désormais est la suivante : pourquoi Google, ayant tout de même démontré la faiblesse de la fonction SHA1, l'utiliserait-elle pour son application de sécurisation de l'authentification ? Plusieurs éléments de réponse sont probables. Tout d'abord, dans le cas du TOTP, compte tenu des performances actuelles des processeurs et de l'informatique actuelle, l'intervalle de 30 secondes pour arriver à casser un OTP reste encore très difficile voire impossible. De plus, à l'inverse des documents PDF, une collision apparente dans un tel mécanisme ne pose pas particulièrement de problème, d'autant plus que la probabilité d'avoir plusieurs conflits à la suite reste faible. Comme il a été démontré en 2005, il faudrait à l'attaquant le temps d'effectuer 2^{63} opérations, c'est-à-dire, en termes de temps processeur, plus de 9000 ans (testé et estimé via un processeur intel Core i7 de première génération). On peut bien entendu penser à une attaque divisant les tâches, ce qui serait beaucoup plus malin de la part de l'attaquant, mais qui, dans le cas des TOTP, ne demeure pas comme une menace de premier plan.

Dans le même ordre d'idées, dans la RFC 4226, il est possible de trouver une partie évoquant la sécurité de l'algorithme SHA1. Dans cette section, il est énoncé que l'on peut considérer SHA1 comme sécurisé pour ce type d'usage dans la mesure où il faudrait autant de temps pour réussir une telle attaque que pour fabriquer un *760-bit RSA modulus*, ce qui est considéré comme infaisable à l'heure actuelle.

Cependant, dans le cas d'un HOTP, n'étant pas limité dans le temps, celui-ci pose alors un problème plus important et les réponses trouvées en faveur de Google s'appauvrissent… D'où le fait, notamment, qu'il est fortement recommandé d'utiliser une authentification par TOTP plutôt qu'une authentification par HOTP. Peut-être serait-il temps pour Google Authenticator d'abandonner sa version HOTP, pour ne laisser place qu'à la

version recommandée et fonctionnelle qu'est le TOTP.

En guise de remédiation, on pourrait suggérer d'intégrer notamment la possibilité pour l'utilisateur de choisir SHA-256, ce qui serait vraiment utile dans le cadre de l'utilisation d'un HOTP. Sur le GitHub du projet, plus particulièrement dans la partie « Issues » [8], une demande (ayant plus d'un an désormais) a été effectuée pour savoir si Google comptait intégrer cette fonctionnalité, ce à quoi un des membres de l'équipe de développement, surnommé ThomasHabets sur le réseau social, a répondu (traduit de l'anglais) :

« *SHA1 est le standard pour HOTP (RFC 4226) mais TOTP (RFC 6238) pourrait utiliser SHA-256. Ce n'est pas en prévision pour le moment, mais les correctifs seraient les bienvenus. Ce n'est pas encore une faille de sécurité pour l'instant. SHA1 n'est pas parfait, mais même avec MD5 « considéré comme cassé » il n'y a pas d'attaque sur la pré-image, donc je ne m'en soucie pas. Cela semble toutefois intéressant, étant donné l'utilisation des tokens SHA256 dont vous parlez. Mais nous aurions besoin de quelqu'un qui a de tels tokens et qui a la foi d'implémenter le code. Je pourrais avoir le dernier, mais pas le premier point.* »

En regardant l'implémentation, on peut malgré tout remarquer qu'en Java (Android), la classe utilisée pour HMAC-SHA1 (`Mac`, de la bibliothèque standard Java) possède des variantes utilisables et sûrement recommandées avec SHA256, SHA384 et SHA512.

```
static Signer getSigningOracle(String secret) {
  try {
    byte[] keyBytes = decodeKey(secret);
    final Mac mac = Mac.getInstance("HMACSHA1");
    mac.init(new SecretKeySpec(keyBytes, ""));

    // Create a signer object out of the standard
    return new Signer() {
      @Override
      public byte[] sign(byte[] data) {
        return mac.doFinal(data);
      }
    };
  } catch (DecodingException error) {
    Log.e(LOCAL_TAG, error.getMessage());
  } catch (NoSuchAlgorithmException error) {
    Log.e(LOCAL_TAG, error.getMessage());
  } catch (InvalidKeyException error) {
    Log.e(LOCAL_TAG, error.getMessage());
  }
```

Figure 6 : Aperçu de la fonction de signature
Source : https://github.com/google/google-authenticator-android/

Cette « maladresse » n'en est certainement pas une, notamment pour des questions de portabilité, et, comme évoqué précédemment, des questions de non-nécessité.

3. b. PIN, partage du secret, et attaques possibles

Parmi toutes les problématiques précédemment soulevées, la taille du PIN reste la victime principale d'une potentielle attaque. Un numéro de 6 chiffres est alors fortement vulnérable aux attaques par force brute. En effet, en considérant un tel numéro, sans compter le temps de latence de transmission des données via le réseau, on tombe sur environ 0.3 secondes de temps d'attaque avec un ordinateur récent, donc avec un processeur Intel Core i5/i7, et environ 8 Go de RAM ou plus. Si l'on imagine que le temps entre l'envoi de la requête et la réponse du serveur a une durée inférieure à 0.001 seconde, alors cela prend

1:30min à l'attaquant de tester toutes les possibilités. Quand on sait que Google Authenticator laisse le choix à l'utilisateur d'agrandir l'intervalle de réponse à 1:30min, cette information fait très peur, encore plus si l'on utilise la fonction HOTP de l'application. Toutefois, une réponse de moins de 0.001 seconde reste assez rare, mais dans le cas de la fonction HOTP, celle-ci reste très vulnérable.

Afin de remédier aux différents problèmes liés aux attaques par force brute et aux attaques de rejeu, Google Authenticator offre de nombreux paramètres réglables côté utilisateur. Lors de la génération du secret, l'application nous propose ainsi (traduits de l'anglais) deux solutions pour pallier ce souci :

- « *Par défaut, les tokens sont valides pendant 30 secondes et dans le but de compenser l'écart de temps entre le client et le serveur, nous autorisons un token avant et après le temps autorisé. Si vous expérimentez des problèmes avec une synchronisation temporelle faible, vous pouvez augmenter la taille par défaut de l'intervalle, de 1:30min à environ 4min. Voulez-vous faire ainsi ?* » D'une manière générale, il est déconseillé de répondre oui à cette question. Cependant, en cas de réel besoin, on peut toutefois autoriser ce mécanisme, mais ce sera une surface d'attaque plus large pour un individu malveillant.

- « *Si l'ordinateur sur lequel vous vous authentifiez n'est pas protégé aux tentatives d'attaque par force brute, vous pouvez activer la limitation de tentatives pour le module d'authentification. Par défaut, ceci limite les attaquants à trois tentatives d'authentification toutes les 30 secondes. Voulez-vous activer la limitation de tentatives ?* » Si l'on n'a pas accès au code de ce mécanisme directement sur le projet open-source, on peut toutefois souligner l'initiative au regard de la sécurité applicative de la solution.

Maintenant, dans la « manière de faire », l'outil Google Authenticator est très critiquable. En effet, le partage se fait par QR Code et/ou par URL, le tout enveloppé par une connexion HTTPS, évitant les *Man in the Middle* et autres attaques possibles. Les QR Codes ont de nombreuses fois démontré leurs faiblesses cryptographiques, c'est pourquoi ont été créés les *Encrypted QR Codes* (EQR), eux-mêmes divisés en deux groupes : les SEQR (*Symmetric EQR*) et les PKEQR (*Public Key EQR*). Un papier [10] de chercheurs en cryptologie d'une université du Massachusetts a été publié il y a quelques temps, expliquant les différents mécanismes et les divers types de QR Codes. On remarque que pour Google Authenticator, ce sont des QR Codes « simples » qui sont utilisés, ce qui ne peut donc pas être considéré comme sécurisé. En plus de cela, on peut également noter que les QR Codes sont parfois gardés en cache dans les navigateurs, ce qui peut laisser une surface d'attaque supplémentaire. Par exemple, sur Google Chrome (au hasard), on peut retrouver en cache les QR Codes dans `chrome://cache`.

Pour en revenir aux attaques *Man in the Middle* mais en se positionnant cette fois au niveau de la validation du token, il faudrait encore que le site utilisant Google Authenticator soit sécurisé. En effet, des attaques de ce type sur des sites WordPress ont montré que l'on pouvait obtenir à la fois le login, le mot de passe, ainsi que le TOTP à l'aide d'une capture de trame; le tout, en clair… Il faut donc prévoir une sécurité de bout en bout.

Cependant, outre les problèmes (parmi tant d'autres) de *shoulder surfing*[2] et autres soucis de *social engineering* liés au fait que les informations apparaissent en clair sur la machine de l'utilisateur, l'aspect que l'on peut critiquer dans les QR Codes autant que dans l'URL est la taille du secret. Celle-ci est de 26 caractères par défaut. En base 32, nous avons,

[2] pratique consistant à observer l'écran d'un individu « par-dessus son épaule » (sans demander son consentement).

sans surprise, 32 possibilités de caractères. Ainsi, on compte $32^{26} \approx 10^{40}$ possibilités.

Une autre attaque, souvent considérée comme triviale, est pourtant fondamentale : le *phone stealing*. Les codes générés par Google Authenticator sont générés et stockés en clair dans l'appareil. Avec des droits root, on peut extraire la base de données content cela. Notamment, sur la version Android, on peut se servir de l'outil *adbd Insecure* pour récupérer les bases de données dans `/data/data/com.google.android.apps.authenticator2/databases/databases`.

peut plus utiliser son application Google Authenticator, mais il n'y en a que cinq, de huit chiffres chacun, ce qui n'avantage pas spécialement l'attaquant, étant donné le nombre de possibilités précédemment cité.

Dernier problème, celui lié au fait que le client et le serveur gardent la même clé. En effet, il s'agit là d'un chiffrement symétrique, ce qui n'est pas forcément sécurisé. Si les serveurs de Google sont compromis, les attaquants ont accès à toutes les informations librement et n'ont pas besoin de clé privée ou quoi que ce soit.

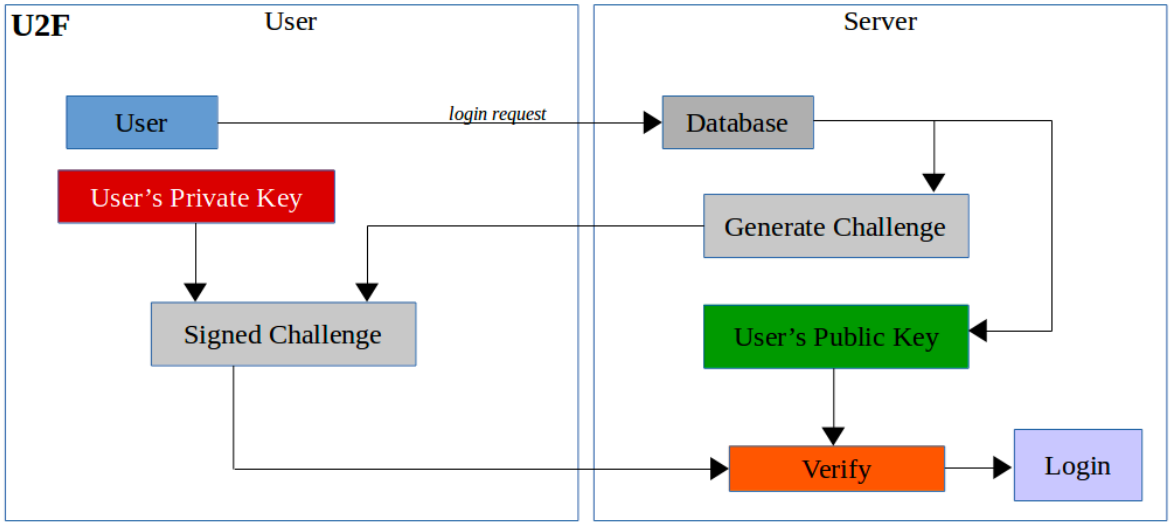

Figure 7 : Schéma du fonctionnement de FIDO/U2F
Source: https://blog.trezor.io

En plus de cela, Google Authenticator a instauré une politique de double-authentification. L'utilisateur possède alors un code de vérification. Quand bien même, ce code de vérification est positionné sur le même écran que le QR Code et l'URL, ce qui biaise le souci de *shoulder surfing* évoqué plus tôt, et donc l'utilité de la double-authentification. Si l'on oublie ce fameux problème, on multiplie notre complexité par 10^6, soit 10^{46} possibilités.

Plusieurs codes de récupération sont disponibles pour l'utilisateur dans le cas où il ne

Une solution a été évoquée dans le blog TREZOR [11], qui relate les différentes faiblesses de Google Authenticator. Il propose alors une solution aux problèmes (que j'ai notamment évoqués ci-dessus), le mécanisme FIDO/UF2. Créé par Google et Yubico, il s'agit d'une solution d'authentification par clé publique visant à pallier les problèmes soulevés par le TOTP. À l'inverse du TOTP, l'utilisateur sera le seul à disposer du secret.

À la fonction utilisée pour la paire de clés près, cette solution contraste grandement

avec le mécanisme TOTP sur le point de vue sécurité.

3. c. Et le temps, dans tout ça ?

Étant donné qu'il s'agit d'un mécanisme traité selon la date courante à la seconde près, il est intéressant de mentionner la problématique de « retard » entre la requête du client et la réponse du serveur. Cette problématique est notamment évoquée dans la RFC 6238. En effet, celle-ci nous annonce que la différence de date (temps sur la machine) est possiblement grande entre le client et le serveur.

Typiquement, lorsqu'un client va envoyer sa requête au serveur, si cette requête est positionnée à la fin des 30 secondes autorisées, il y a de fortes chances pour que le moment de la réception ne soit pas dans les mêmes 30 secondes que celles de la requête. Ainsi, il pourrait être utile pour le serveur d'avoir la possibilité de comparer les OTP générés dans un certain intervalle pour voir s'ils correspondent aux requêtes du client. Cependant, cela laisse une plus grande surface pour les potentiels attaquants. Ainsi, la RFC recommande un intervalle de 30 secondes car c'est un « juste-milieu entre l'utilisabilité et la sécurité ». De plus, côté utilisateur, si celui-ci souhaite s'authentifier sur deux appareils différents avec la même session et donc le même générateur de TOTP, il doit attendre un certain temps. En effet, deux authentifications à succès à la suite ne sont pas possibles dans le même intervalle de 30 secondes. Ainsi, 30 secondes semble la bonne durée. En effet, une attente de 5 minutes pour un utilisateur n'est pas acceptable.

De plus, pour des problématiques de longévité de la solution et de la compatibilité avec de nombreux appareils, notamment embarqués, il faudra, en accord avec la RFC 6238, que l'implémentation de l'algorithme TOTP supporte une valeur de T supérieure à $2^{31} - 1$. Cette proposition fait echo au « bug de l'an 2038 », qui consiste à ce que les représentations de la date sur 32 bits soient obsolètes en l'an 2038. Cela est dû au fait que la date est en fait un nombre de secondes depuis la date « *epoch* » évoquée avant, qui dépassera la taille de 32 bits et rendra donc les dates fausses.

Enfin, il est nécessaire de mentionner des attaques possibles de Google Authenticator se basant sur la métrique temporelle. En particulier, le trafic NTP non-authentifié présente plusieurs risques. Un papier [22] détaille notamment de nombreuses façons d'attaquer le protocole NTP. En combinant plusieurs prérequis, un attaquant peut potentiellement prendre le contrôle de l'horloge d'un serveur distant et donc rendre les TOTP obsolètes. Cependant, les démons NTP tels que ntpd n'acceptent les *timestamps* issus de serveurs NTP que sous certaines conditions, il y a des algorithmes de validation. On peut toutefois penser à une attaque poussée dans laquelle d'autres mécanismes exploiteraient des vulnérabilités sur ntpd.

4. Conclusion

Par sa facilité d'installation, d'utilisation et d'intégration à divers modules tels que PAM ou encore des systèmes d'authentification web comme celui de Facebook, Google Authenticator a gagné de nombreux utilisateurs en plusieurs années.

Cet outil permet de générer des TOTP selon l'algorithme défini dans la RFC 6238. Il s'agit en fait d'un dérivé des HOTP. Un HOTP est basé sur le principe des HMAC, c'est-à-dire une combinaison de hachages, réalisés en l'occurrence par SHA1, à partir d'un secret issu d'un événement aléatoire. Diverses sources d'aléatoire existent, aussi bien logicielles que matérielles. Celles-ci sont reconnues grâce à des tests officiels qui permettent de tester si l'on peut ou non considérer ces sources comme aléatoires.

Si Google Authenticator propose également de générer des HOTP, cette fonctionnalité reste toutefois à proscrire pour ses nombreux défauts de sécurité et sa facilité d'attaque. D'un point de vue sécuritaire, il est donc conseillé de privilégier TOTP face à HOTP, et fortement déconseillé de l'utiliser tout de même comme seul facteur d'authentification.

Le fonctionnement de Google Authenticator se résume en trois grandes étapes. La première est la génération du secret, son encodage en base32 et son partage entre le client et le serveur. Il s'agit de l'étape la plus importante, car si ce secret est compromis, le reste des étapes demeurera obsolète. La seconde étape correspond à la génération d'OTP, c'est-à-dire la signature via HMAC-SHA1, ensuite rogné pour n'avoir que six chiffres. Enfin, la troisième étape correspond au décodage des messages du client par le serveur, dépendant bien entendu de la synchronisation temporelle entre les deux entités.

Comme tout mécanisme, celui-ci possède des défauts, tant au niveau de la réalisation, que dans la qualité cryptographique de la solution. À première vue, l'utilisation de SHA1 comme fonction de hachage pour HMAC paraît absurde, surtout venant de Google, étant donné que des chercheurs (parmi d'autres) du fameux géant de la Silicon Valley ont démontré il y a deux ans maintenant l'obsolescence de l'algorithme en question. Toutefois, les besoins cryptographiques d'un tel outil ne requièrent pas des algorithmes plus robustes pour le moment.

En revanche, pour ce qui est du *token* unique, les six chiffres restent d'une complexité assez pauvre, jouant toutefois sur leur durée éphémère. Concernant la diffusion du secret et sa détention, le débat de la sécurité des URL/QR Codes revient souvent sur le tapis des opposants à Google Authenticator. De même, le mécanisme de secret symétrique entre le client et le serveur est très controversé, certains proposant même de nouvelles solutions comme FIDO/U2F, une variante à clé publique de TOTP. Un des principaux problèmes rencontrés par les implémentations de TOTP reste la gestion du temps, évoquée brièvement dans la RFC. Celle-ci nous conseille de rester à 30 secondes, ce qui nous donne un bon compromis entre sécurité et exploitabilité par les utilisateurs.

Tous ces principes de sécurité liés au niveau cryptographique de la solution dépendent directement des progrès de l'informatique actuelle, en particulier de la vitesse des processeurs et des échanges clients-serveurs. Dans un futur proche, une technologie comme les ordinateurs quantiques et leur rapidité de calcul décuplée pourrait faire changer un très grand nombre de principes de la cryptologie jusqu'alors considérés comme sécurisés.

Bibliographie

[1] *Google Authenticator*,
https://fr.wikipedia.org/wiki/Google_Authenticator

[2] *RFC 6238, TOTP : Time-Based One-Time Password Algorithm*,
D. M'Raihi, S. Machani, M. Pei et J. Rydell
Mai 2011
https://tools.ietf.org/html/rfc6238

[3] *RFC 4226, HOTP: An HMAC-Based One-Time Password Algorithm*,
D. M'Raihi, M. Bellare, F. Hoornaert, D. Naccache et O. Ranen
Décembre 2005
https://tools.ietf.org/html/rfc4226

[4] *RFC 2104, HMAC: Keyed-Hashing for Message Authentication*,
H. Krawczyk, M. Bellare et R. Canetti
Février 1997
https://tools.ietf.org/html/rfc2104

[5] *RFC 3174, US Secure Algorithme 1 (SHA1)*,
D. Eastlake and P. Jones
Septembre 2001
https://tools.ietf.org/html/rfc3174

[6] *Google Authenticator Project*,
Google Development Team (ThomasHabets, GrayTShirt, klyubin, and others)
Dernières modifications du code en 2016
https://github.com/google/google-authenticator

[7] *How Google Authenticator works?*,
14 Septembre 2014
https://garbagecollected.org/2014/09/14/how-google-authenticator-works/

[8] *Google Authenticator Libpam Project*, *Issues section*,
ThomasHabets
9 novembre 2016
https://github.com/google/google-authenticator-libpam/issues/11

[9] *Software Random Number Generation Based on Race Conditions*,
Adrian Coleșa, Radu Tudoran et Sebastian Banescu
Octobre 2008
https://www.researchgate.net/publication/
224578073_Software_Random_Number_Generation_Based_on_Race_Conditions

[10] *Security Overview of QR Codes*,

Kevin Peng, Harry Sanabria, Derek Wu and Charlotte Zhu
https://courses.csail.mit.edu/6.857/2014/files/12-peng-sanabria-wu-zhu-qr-codes.pdf

[11] *Why You Should Never Use Google Authenticator Again*,
SatoshiLabs
28 octobre 2016
https://blog.trezor.io/why-you-should-never-use-google-authenticator-again-e166d09d4324

[12] *How does Google Authenticator Work ?*
Réponses de Arjun SK et Ayrx
Dernières modifications le 4 novembre 2013 et le 20 avril 2017
https://security.stackexchange.com/questions/35157/how-does-google-authenticator-work

[13] *Google Authenticator Wiki*,
Thomas Habets
Dernière modification le 4 juillet 2017
https://github.com/google/google-authenticator/wiki

[14] *Implementing Google's Two-Step Authentication to Your App*,
Vyacheslav
Dernière modification le 12 avril 2017
https://www.codementor.io/slavko/google-two-step-authentication-otp-generation-du1082vho

[15] *Google Authenticator implementation in Python*,
Réponse de Tadeck
Dernière modification le 13 Novembre 2012
https://stackoverflow.com/questions/8529265/google-authenticator-implementation-in-python

[16] *SHA1*,
https://fr.wikipedia.org/wiki/SHA-1

[17] *How does HOTP keep in sync?*,
Réponse de Thomas Pornin
28 Septembre 2011
https://crypto.stackexchange.com/questions/839/how-does-hotp-keep-in-sync

[18] *Base32,*
https://en.wikipedia.org/wiki/Base32

[19] *SHAttered : Google a cassé la fonction de hachage SHA-1*,
Julien Cadot
23 février 2017
https://www.numerama.com/tech/235436-shattered-google-a-casse-la-methode-de-chiffrement-sha-1.html

[20] Tricking Google Authenticator TOTP with NTP
Gabor
https://blog.gaborszathmari.me/2015/11/11/tricking-google-authenticator-totp-with-ntp/

[21] Code source du projet Delorean
PentesterES
https://github.com/PentesterES/Delorean

[22] Attacking the Network Time Protocol
Aanchal Malhotra, Isaac E. Cohen, Erik Brakke, et Sharon Goldberg
20 août 2015
http://www.cs.bu.edu/~goldbe/papers/NTPattack.pdf

Annexes

```java
protected String encodeInternal(byte[] data) {
  if (data.length == 0) {
    return "";
  }

  // SHIFT is the number of bits per output character, so the length of the
  // output is the length of the input multiplied by 8/SHIFT, rounded up.
  if (data.length >= (1 << 28)) {
    // The computation below will fail, so don't do it.
    throw new IllegalArgumentException();
  }

  int outputLength = (data.length * 8 + SHIFT - 1) / SHIFT;
  StringBuilder result = new StringBuilder(outputLength);

  int buffer = data[0];
  int next = 1;
  int bitsLeft = 8;
  while (bitsLeft > 0 || next < data.length) {
    if (bitsLeft < SHIFT) {
      if (next < data.length) {
        buffer <<= 8;
        buffer |= (data[next++] & 0xff);
        bitsLeft += 8;
      } else {
        int pad = SHIFT - bitsLeft;
        buffer <<= pad;
        bitsLeft += pad;
      }
    }
    int index = MASK & (buffer >> (bitsLeft - SHIFT));
    bitsLeft -= SHIFT;
    result.append(DIGITS[index]);
  }
  return result.toString();
}
```

Fonction d'encodage en base32 dans la version Android (Java) de l'application
Source : https://github.com/google/google-authenticator-android/