



SECURITÉ

ONE TIME PASSWORD

GROUPE MOHAMMED **ROUABAH**, WILLIAM **MAILLARD**



25/01/2023

Table des matières

	Abstract	2
	Introduction	3
1	Exemple d'utilisation d'OTP dans le 2FA	4
1.1	Activation du 2FA : partage de secret entre le client et le serveur	4
1.2	Génération et variantes d'OTP	5
1.2.1	HOTP	5
1.2.2	TOTP	6
1.3	Gestion de la désynchronisation dans la génération des OTP entre le client et le serveur ...	7
1.4	Utilisation d'OTP dans une authentification bi-directionnelle	7
2	Algorithme de génération d'OTP	9
2.1	Gestion de la seed	9
2.1.1	Création	9
2.1.2	Communication	10
2.1.3	Stockage de la seed et conseils de sécurité	10
2.2	Choix d'une donnée incrémentale	10
2.3	Utilisation de HMAC	10
2.3.1	Définition	10
2.3.2	Exemple d'utilisation concret	11
2.4	Troncation de la sortie du HMAC	12
2.4.1	Définition	12
2.4.2	Exemple d'utilisation concret	12
3	Prototype python de l'algorithme OTP	13
3.1	Architecture et déploiement	13
3.2	Utilisation de pyotp	13
3.3	Implémentation personnelle de l'algorithme OTP	14
3.4	Mise en place du 2FA dans une application	17
4	Conclusion : failles de sécurité et bonnes pratiques	19
A	Vocabulaire	i
B	Références	iii
C	Illustrations du prototype	v

Résumé

Ce rapport introduit les différents concepts inhérents à l'algorithme OTP en prenant l'exemple de son utilisation dans le protocole 2FA. Ensuite il explique en détaille et en déroulant un exemple concret, les différentes partie de l'algorithme permettant de générer un OTP à partir d'une clé privé partagé et d'une donnée incrémentale. Puis il montre la mise en place du 2FA avec les deux variantes d'OTP , HOTP et TOTP , sur un prototype réalisé en python. Finalement, ce rapport conclus en mettant en exergue les failles de sécurité possible à l'utilisation d'OTP et quelles sont les bonnes pratiques à mettre en place pour s'en prémunir.

Introduction

Un **mot de passe à usage unique**, désigné sous l'acronyme anglais **OTP** dans la suite de ce rapport, est une séquence d'au moins **six chiffres** ou caractère généré à partir d'une **clé privé** et d'une **donnée itérative** afin de **valider une action** utilisateur comme par exemple une authentification ou une transaction bancaire.

Ces codes sont soit générés avec une **application** dite 'authenticator' soit **envoyé** à l'utilisateur cherchant à valider une action par courriel ou message. Ces codes sont donc générés par un moyen que **seul l'utilisateur concerné possède**.

Les OTP sont donc tout naturellement utilisés dans les méthodes d'authentification dites '**2FA**' qui consiste à vérifier que l'utilisateur **possède quelque chose** en plus de connaître son mot de passe. Plus particulièrement cela permet d'attester que l'utilisateur possède **un appareil physique** (téléphone mobile ou clé otp) ou **un compte virtuel** (mail ou numéro de téléphone).

D'autre part, l'utilisation des OTP ne se limitent pas au 2FA, mais est aussi **utilisé dans de nombreux domaines** comme par exemple :

Systèmes de Port Knocking : Ils utilisent des OTP pour ajouter une couche de sécurité à l'accès réseau. L'utilisateur génère un OTP valide en tapant une séquence spécifique de requêtes vers des ports prédéfinis, permettant ainsi l'ouverture temporaire de ports et l'accès au réseau.

Preuve de Localisation Préservant la Vie Privée : Dans ce contexte, les OTP peuvent être utilisés pour valider la présence d'un individu à un endroit particulier sans révéler de données sensibles. L'individu génère un OTP qui atteste de sa localisation actuelle sans divulguer d'informations précises.

VPN d'Entreprise : Des clés OTP permettant de générer des TOTP sont utilisés dans les VPN d'entreprise pour renforcer l'authentification. Les employés utilisent ces codes pour générer des codes valides, nécessaires pour établir une connexion sécurisée au réseau de l'entreprise.

Validations Bancaires et de Paiement Électronique : Les OTP sont couramment utilisés dans les validations bancaires et les transactions de paiement électronique. Lorsqu'un utilisateur effectue une transaction, un OTP est généré et envoyé à son appareil mobile, assurant une couche supplémentaire de sécurité en vérifiant l'authenticité de la transaction.

Ainsi, nous allons étudier, dans ce rapport, les **mécanismes cryptographiques** permettant de générer ces codes à usage unique.

Pour cela nous analyserons **l'utilisation des OTP dans le protocole 2FA** afin d'en extraire des **concepts** inhérents et comprendre la divergence entre les **deux variantes** que constituent l'HOTP et le TOTP.

Ensuite, à l'aide des concepts définis précédemment, nous construirons un **algorithme de génération d'OTP**, en veillant à comprendre les **enjeux de sécurité** derrière chaque **choix de paramètre** de cet algorithme.

Puis nous expliquerons comment **intégrer une validation OTP** dans son application en s'appuyant sur le **prototype python** qui a été développé dans le cadre de cette étude.

Finalement, nous expliquerons les **failles de sécurité** que peut présenter l'utilisation d'OTP afin de terminer sur les **points de sécurité conseillés** afin d'assurer une utilisation sécurisée de cette technologie.

1 Exemple d'utilisation d'OTP dans le 2FA

Comme expliqué dans l'introduction les OTP sont utilisés dans le [protocole de double authentification](#) 2FA visant à vérifier que [l'utilisateur possède quelque chose](#) en plus de vérifier la connaissance de son mot de passe.

Dans cette partie nous allons ainsi nous intéresser aux [application 'authenticator'](#) permettant de générer des OTP en étudiant dans un premier temps quels sont les [mécanismes](#) à mettre en place [pour activer le 2FA](#).


Puis, dans un second temps, nous verrons comment l'OTP permet de [valider l'authentification](#) de l'utilisateur dans le contexte du 2FA, et [les variantes](#) que constituent [HOTP](#) et [TOTP](#).


Et enfin nous déduirons de ces exemples les paramètres à prendre en compte pour palier les [désynchronisation de génération d'OTP](#) entre le client et le serveur.

1.1 Activation du 2FA : partage de secret entre le client et le serveur

Afin d'attester de l'identité d'un client avec un code OTP, il faut que chaque partie (le client et le serveur) possède [un moyen de générer des codes identiques](#). Ainsi lorsque l'utilisateur active le 2FA sur son compte le serveur va créer et [communiquer une clé secrète](#) au client avant de lui même [stocker](#) cette valeur. Cette clé va servir de [graine](#), ou seed en anglais, à la [génération d'OTP identiques](#) du côté du client et du serveur permettant à ce dernier d'attester l'identité du premier.

Le [partage](#) de cette clé secrète est réalisé à travers la génération d'un [QRcode](#) par le serveur et le scanne de ce dernier par le client utilisant une application 'authenticator' comme freeOTP. L'exemple ci-dessous montre le contenu d'un exemple de QRcode généré pour activer le 2FA d'un compte github.



 Decode Succeeded

https://zxing.org/w/decode.jspx

Raw text	otpauth://totp/GitHub:gabzim60secret=27b4dakb6h3fuxco&issuer=GitHub
Raw bytes	44 46 f7 47 06 17 57 46 83 a2 f2 f7 46 f7 47 02 f4 76 97 44 87 56 23 a6 76 16 27 a6 96 d3 63 03 f7 36 56 37 26 57 43 d3 23 76 23 46 46 16 b6 23 66 83 36 67 57 86 36 f2 66 97 37 37 56 57 23 d4 76 97 44 87 56 20 ec 11 ec 11 ec 11 ec 11 ec 11 ec 11 ec 11 ec 11
Barcode format	QR_CODE
Parsed Result Type	URI
Parsed Result	otpauth://totp/GitHub:gabzim60?secret=27b4dakb6h3fuxco&issuer=GitHub

FIGURE 1 – Exemple de QRcode contenant une seed partagée par le serveur

Nous pouvons donc en déduire qu'une [url de partage de seed](#) pour l'algorithme OTP est constitué des éléments suivants :

- un **protocole** : [otpauth](#) ://
- la **variante d'OTP** utilisée : /totp ou /hotp (expliqué dans la partie suivante)
- l'**émetteur** de la seed : dans l'exemple /GitHub :nomUtilisateur
- **deux paramètres** :
 1. le **secret** encodé en base32 : secret=27b4dakb6h3fuxco
 2. le nom de l'**émetteur** : issuer=Github

1.2 Génération et variantes d'OTP

Maintenant que le client et le serveur possèdent une **donnée commune**, ils leur manquent un **moyen de créer des codes à usage unique** à partir de cette clé privée. Cela est le rôle de la **donnée incrémentale**. La valeur de cette donnée dépend de la variante d'OTP choisie.

1.2.1 HOTP

La première variante d'OTP est appelé HOTP qui est l'acronyme de '**HMAC -based OTP**' et qui se traduit par 'OTP basé sur HMAC', et utilise un **compteur** comme donnée incrémentale.

En effet, à chaque fois que le client génère un OTP il incrémente un compteur, et le serveur fait de même à la réception d'un OTP valide. Le schéma ci-dessous illustre le mécanisme d'authentification en utilisant HOTP.

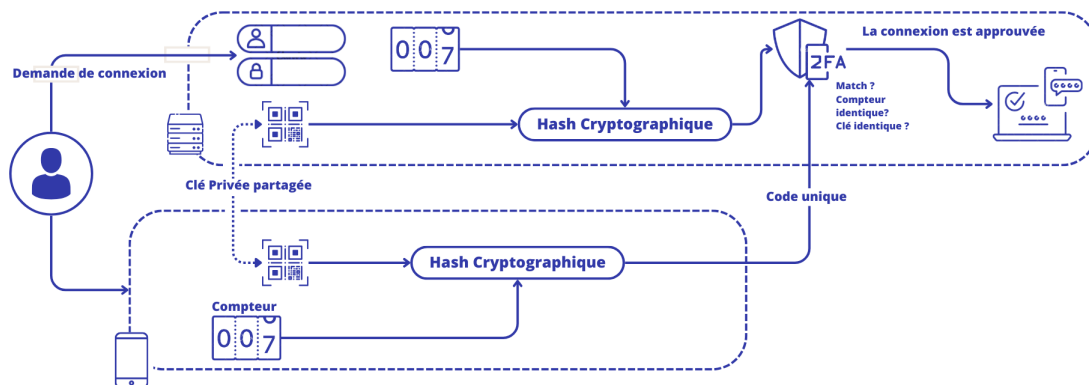


FIGURE 2 – Exemple 2FA : HOTP

Ainsi l'algorithme HOTP nécessite une **synchronisation constante du compteur** entre le client et le serveur. Néanmoins une **désynchronisation** peut se produire du fait que le client incrémente le compteur à chaque fois qu'il veut créer un code sans savoir s'il a été accepté par le serveur, c'est-à-dire que le serveur à lui aussi incrémenter son compteur.

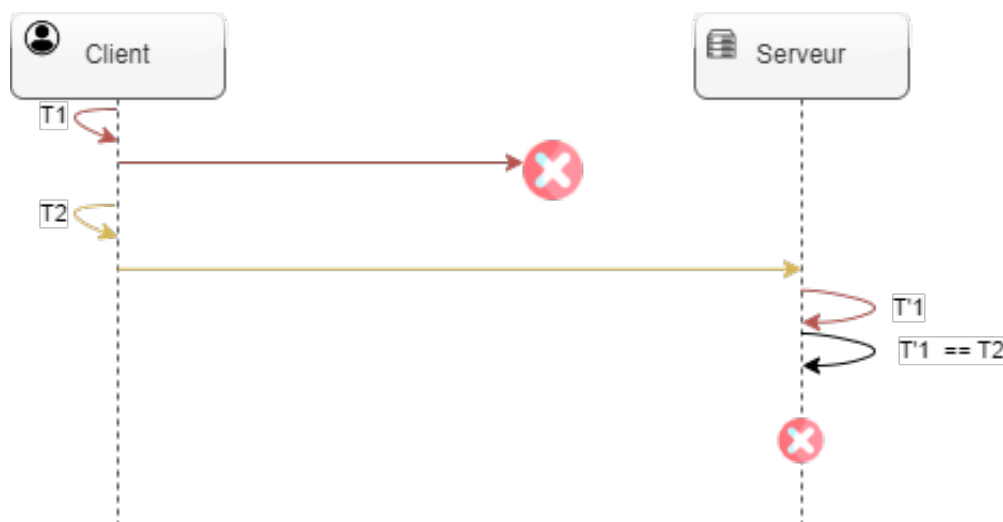


FIGURE 3 – HOTP : exemple de désynchronisation

Pour palier à ce problème, on peut définir un **paramètre 'look ahead window'** ('fenêtre d'anticipation' en français) qui permet d'ajuster **le nombre s de prochaines valeurs** que peut vérifier le serveur afin de **détecter les cas de désynchronisation** lorsque le code reçu ne correspond pas au code courant généré.

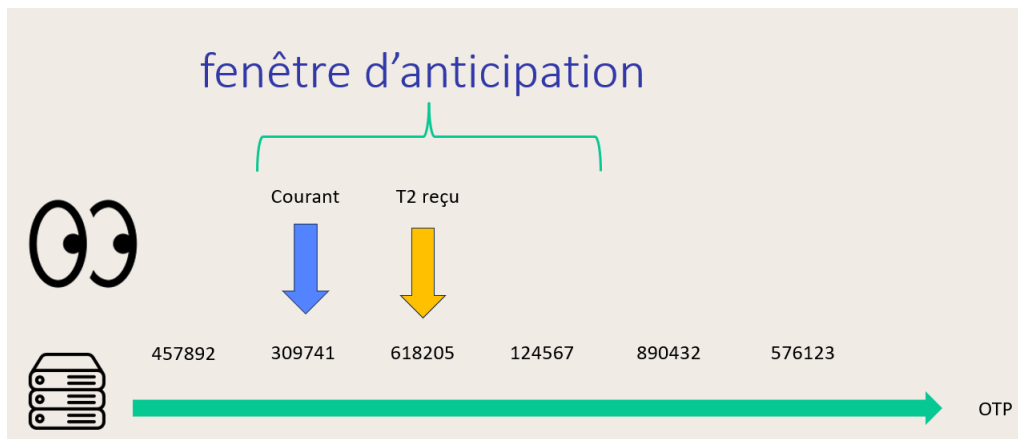


FIGURE 4 – paramètre HOTP : look ahead window

1.2.2 TOTP

La deuxième variante d'OTP est appelé TOTP qui est l'acronyme de 'Time-based OTP' et qui se traduit par 'OTP basé sur le temps'. Il utilise donc un timestamp, plus particulièrement le temps 'UNIX' correspondant au nombre de secondes écoulé depuis le premier janvier 1970, comme donnée incrémentale.

Le temps UNIX a été choisi car il est disponible sur la grande majorité des appareils qui tournent sous linux, et en particulier les téléphones mobiles android et apple qui sont utilisés dans la plupart des cas.

Le délai temporel nécessaire pour générer un nouveau code est généralement fixé à 30 secondes et commence au début d'une nouvelle minute précise. Par exemple, pour les instants de génération, le premier code est associé à 00 :00, le deuxième à 00 :30, le troisième à 01 :00, et ainsi de suite. Ainsi, lorsque l'algorithme doit générer un nouveau code, il arrondit le temps vers le bas jusqu'à l'instant le plus proche marqué par les secondes 00 ou 30.

De ce fait le client et le serveur utilisent le temps courant de leur système arrondis afin de générer un OTP, comme l'illustre le schéma ci-dessous

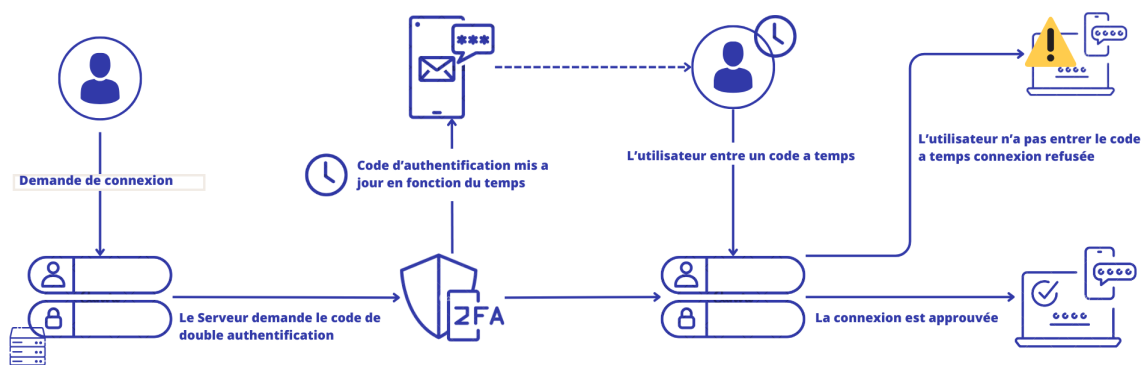


FIGURE 5 – Exemple 2FA : TOTP

Néanmoins, certains appareils physiques (en particulier dans le domaine des iot) n'ayant pas accès à l'heure d'internet subissent un décalage estimé de deux minutes par an sans possibilité de réajuster le temps. Il peut ainsi être pertinent de configurer un paramètre 'look ahead window' pour l'utilisation de TOTP dans certains cas spécifiques.

Un autre problème rencontré avec l'utilisation de TOTP est la latence du réseau. En effet, un code valide à son envoi n'est pas garanti d'être valide à son arrivé au serveur suivant le temps mis pour y parvenir, comme illustré dans le schéma ci-dessous.

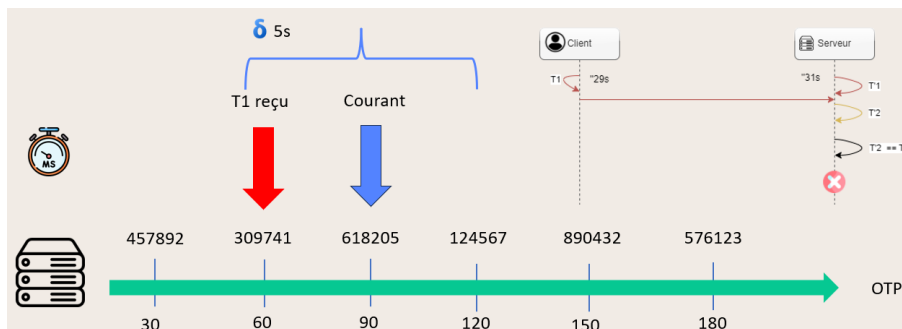


FIGURE 6 – TOTP : problème de la latence réseau

Ainsi on peut définir une **marge temporelle** afin que le serveur compare aussi le code OTP reçu avec le code OTP précédent qui rentre dans la marge temporelle, afin de rendre l'utilisation du protocole d'authentification plus fluide pour le client.

1.3 Gestion de la désynchronisation dans la génération des OTP entre le client et le serveur

Une fois que le serveur a **découvert une désynchronisation** du OTP client reçu avec ceux qu'il a générés dans le 'look-ahead window', il va aligner son compteur en mémoire avec celui du client et **enclencher le protocole de resynchronisation** qui consiste à **demandeur n OTP consécutifs** afin de s'assurer que le client possède bien la clé secrète et que ce n'était pas une déduction du code.

Augmenter n va permettre de réduire le succès des attaques de type de brute-force, mais va rallonger le processus de resynchronisation. Ce paramètre n est **en général fixé à 3**.

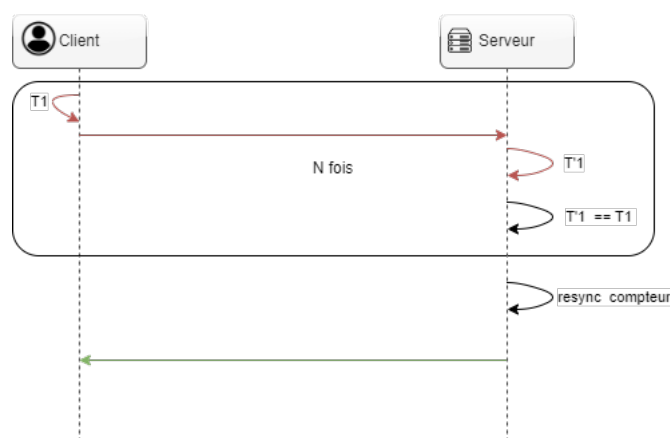


FIGURE 7 – HOTP : resynchronisation du compteur entre le client et le serveur

Finalement, si la **resynchronisation est un succès**, c'est à dire les n codes générés par le client ont été validés par le client, le serveur peut **mettre à jour son compteur** par rapport au dernier OTP reçu.

Dans le cas où la **resynchronisation est un échec** le serveur ne met pas son compteur à jour et **attend de nouveau un OTP** jusqu'à ce que la limite fixée par le **paramètre d'étranglement soit atteinte**.

Une fois cette limite atteinte, le serveur n'accepte plus d'OTP pour ce compte et le bloque avant d'en notifier le propriétaire qu'une activité suspecte de connexion a été détectée.

1.4 Utilisation d'OTP dans une authentification bi-directionnelle

Il est intéressant de noter que l'utilisation d'OTP peut aussi permettre d'effectuer une **authentification bidirectionnelle** permettant au client et au serveur de **vérifier leurs identités mutuellement**.

En effet, en vérifiant chacun leurs tour un OTP reçue de l'autre partie, ils vont chacun pouvoir attester de la connaissance de la clé privé par chacun.

Ce protocole d'authentification bi-directionnelle se compose donc des étapes suivantes (en supposant que le client et le serveur ne présente pas de problèmes de synchronisation traités ci-dessus) :

1. Le client génère **OTP-C1** et l'envoie au serveur
2. Le serveur reçoit **OTP-C1** et génère **OPT-S1**,
3. Le serveur compare **OPT-C1** et **OPT-S1** :
 - si égalité, alors génère **OPT-S2** et l'envoie au client,
 - sinon stop le processus.
4. Le client reçoit **OPT-S2** et génère **OPT-C2**,
5. Le client compare **OPT-C2** et **OPT-S2** :
 - si égalité, alors vérification terminée, le client utilise le serveur,
 - sinon, le client stop le processus.



FIGURE 8 – protocole d'authentification bi-directionnelle basé sur OTP

2 Algorithme de génération d'OTP

L'algorithme de génération d'OTP se compose des quatre étapes suivantes :

1. La gestion d'une '**seed**' par client permettant de générer des codes pseudo-aléatoires,
2. Le choix d'une **donnée incrémentale**,
3. La paramétrisation d'une **méthode de cryptographie** nommé HMAC ,
4. La **troncation** de la sortie du HMAC pour obtenir un code à la longueur souhaité.

Chacune des sous parties de cette section traitera donc d'une partie de cet algorithme afin d'en détailler tous les rouages et de comprendre les enjeux de sécurité liés à chacun d'entre eux.

2.1 Gestion de la seed

La seed est une donnée privée connu uniquement du client et du serveur dont la **création**, la **communication** et le **stockage** vont être des points critiques à la **sécurité de l'algorithme**. Il est conseillé de prendre une **clé de même taille que la sortie de la fonction de hachage** utilisée par la méthode HMAC pour garantir l'**interopérabilité** entre les systèmes, c'est à dire utilisé une taille fixe qui ne dépend pas de la représentation d'un type (int, float, char, ...) faite par un système et ainsi limiter les problèmes de compatibilité sous-jacent.

Voici des exemple de taille de clé conseillé selon la fonction de hachage donnée en paramètre de la méthode HMAC :

SHA-1 160 bits

SHA-256 256 bits

SHA-512 512 bits

2.1.1 Création

Pour générer la seed on a le choix entre deux méthodes :

🎲 la génération **aléatoire**

🎲 la génération **déterministe**

Génération aléatoire

Cette première méthode est divisée en 2 catégories selon les outils utilisés pour la génération de l'aléatoire.

Ainsi les générateurs s'appuyant sur l'aléatoire d'un **phénomène physique** comme par exemple un oscillateur ou un capteur environnemental sont désignés sous le terme '**hardware-based generator**'.

Tandis que les générateurs basés sur des **algorithmes de pseudo-aléatoire** comme le LFRS (Linear Feedback Shift Register) sont désignés sous le terme '**software-based generator**'.

Génération déterministe

La deuxième méthode est elle aussi divisé en deux catégories.

Soit on va utiliser une **unique Master Key (MK)** générée avec une méthode évoquée précédemment afin d'en **dérivée une seed** pour chaque client, en utilisant une **donnée publique** de ce dernier comme par exemple un numéro de série (*i*). La seed est ensuite calculé en faisant un XOR entre la master key et la donnée client, puis en calculant le hash de cette valeur.

$$k_i = SHA-1(MK \oplus i)$$

Soit on utilise une **Master Key par client (MK_i)** et on utilise également une donnée *j* du client pour **dérivée une seed**. Chaque appareil est donc associé à un couple (*i, j*) afin de retrouver la seed associée.

$$k_{ij} = SHA-1(MK_i \oplus j)$$

2.1.2 Communication

En ce qui concerne la [transmission de la seed](#) du serveur au client, il est impératif d'opter pour un [canal de communication sécurisé](#) tel que [TLS/SSL](#). Ce qui est souvent le cas, car OTP est souvent utilisé pour valider une action d'un client ayant déjà établie une connexion sécurisée.

2.1.3 Stockage de la seed et conseils de sécurité

Maintenant, en ayant pris connaissance de ces différentes méthodes, on peut se demander laquelle est plus sécurisée ? Cela dépend en fait du besoin.

En effet, si l'objectif est de [maximiser la sécurité de la clé](#), une approche basée sur le matériel ([hardware-based](#)) est préférable, car elle est [difficilement prédictible](#). Cependant, cette assertion pourrait être remise en question avec l'émergence des ordinateurs quantiques, capables de modéliser l'aspect quantique des phénomènes réels...

Pour une [méthode déterministe](#), l'utilisation d'[une clé maître par client est préférable](#), car si une clé est compromise, cela ne compromet pas l'ensemble des seeds des clients.

D'un autre côté, si l'objectif est de [maximiser la sécurité du stockage](#) de la clé, il est préférable d'adopter une [méthode avec une ou plusieurs clés maîtres](#). Cela [évite le stockage direct de la seed](#) sur le serveur, contrairement à la méthode aléatoire, qui nécessite le stockage de la clé chiffré par un algorithme symétrique car sa génération n'est pas reproductible.

En ce qui concerne le [stockage de la clé](#) avec un algorithme cryptographique symétrique l'[AES](#) est fréquemment utilisé.

Pour **renforcer la sécurité**, plusieurs stratégies peuvent être adoptées :

- Utilisation de la [méthode de Shamir](#) : Diviser la clé en plusieurs morceaux et les stocker de manière indépendante, éventuellement sur des serveurs distincts. Cela augmente la sécurité du stockage en rendant la récupération complète de la clé plus complexe.
- Utilisation de [données difficiles à obtenir](#) pour la clé du client : Intégrer des informations telles que le [mot de passe chiffré de l'utilisateur](#), son numéro de téléphone, ou un calcul d'identifiant basé sur plusieurs données de l'utilisateur. Ceci crée un "secret partagé composite" (composite-shared secret), augmentant la complexité pour un attaquant de compromettre la sécurité.

2.2 Choix d'une donnée incrémentale

Le choix entre une donnée incrémentale [basée sur le temps](#), comme le TOTP, ou sur [un compteur](#), comme le HOTP, dépend des exigences spécifiques de chaque application. Dans des contextes où la [synchronisation temporelle est critique](#), le [TOTP](#) est avantageux, garantissant la validité temporelle des codes. À l'inverse, lorsque [la séquentialité des actions est plus pertinente](#), le [HOTP](#) est préférable, générant des codes uniques en fonction d'un compteur indépendamment du temps. Ainsi, la décision repose sur les caractéristiques particulières du système, les contraintes temporelles et [les besoins de sécurité spécifiques à chaque scénario d'utilisation](#).

2.3 Utilisation de HMAC

2.3.1 Définition

Avec les deux données précédentes ([seed](#), [compteur](#)), une [fonction cryptographique](#) appelée [HMAC](#) est employée pour calculer un hash à l'aide d'une fonction telle que SHA-256. Contrairement à un simple hachage XOR du message et de la clé, HMAC [modifie ces deux valeurs avant de les hacher](#).

Voici le **processus général** de la fonction HMAC :

1. Dans un premier temps, on compare la **taille de la clé** avec la **taille du bloc d'entrée** de la fonction de hachage, et
 - (a) Si elle est plus grande, on la hache afin d'obtenir une clé de la bonne taille.
 - (b) Si elle est plus petite, on la complète par des zéros.

Cela nous donne la clé effective.

2. On va ensuite XORer la clé effective avec l'*iPad* (aussi appelé **constante de remplissage interne**), qui est une séquence binaire de la même taille donnée en paramètre de HMAC (souvent 0x36 répété x fois pour remplir le bloc), ce qui donne R_0 .
3. Ensuite, on **concatène** le message à la fin de R_0 pour obtenir R_1 . Puis, on applique la fonction de **hachage** sur le résultat précédent pour obtenir $H - 1$.
4. Ensuite, on XORe la clé effective avec l'*oPad* (aussi appelé **constante de remplissage externe**, souvent 0x5C), ce qui donne R_2 .
5. On **concatène** ensuite R_2 et H_1 . Et enfin, on applique la fonction de **hachage** sur ce résultat pour obtenir le résultat du HMAC .

(NB : ici, on utilise une fonction de **hachage par bloc** et non linéaire)

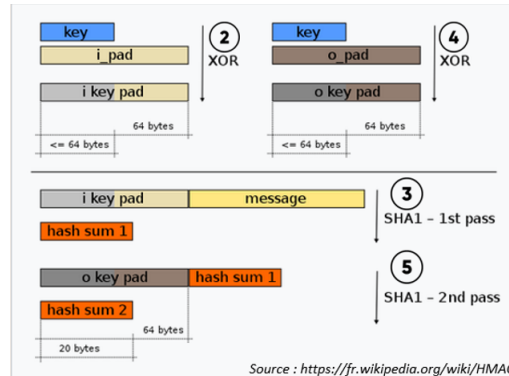


FIGURE 9 – schéma de la page wikipédia de hmac

2.3.2 Exemple d'utilisation concret

Pour des raisons de clarté, nous ferons les **hypothèses suivantes** dans notre exemple :

- La taille du bloc de données est de 8 bits.
- La taille de la clé est de 16 bits.
- Les constantes *iPad* et *oPad* sont respectivement 0x36 et 0x5c, répétées pour remplir le bloc.

Supposons une clé effective de 0xABCD, un compteur de 0x1234, et un message de 0x5678. En utilisant SHA-256 comme fonction de hachage, les constantes *iPad* et *oPad* de 0x36 et 0x5C respectivement, le processus serait le suivant :

1. **Ajustement de la Clé** : La clé effective reste 0xABCD.
2. **Calcul de R_0** : $R_0 = \text{clé} \oplus \text{iPad} = 0xABCD \oplus 0x36363636 = 0x36369dfb$.
3. **Concaténation et Hashing** : $h_1 = \text{SHA-256}(0x36369dfb5678) = 0xbf72f9508c996418e086a5cb96e2399c92fde8a8dde7e1199fad265cf7323d3a$
4. **Calcul de R_2** : $R_2 = \text{clé} \oplus \text{oPad} = 0xABCD \oplus 0x5C5C5C5C = 0x5c5cf791$.
5. **Finalisation du HMAC** : Concaténation de R_2 et H_1 , puis application de SHA-256 pour obtenir le résultat final du HMAC :

$$\text{HMAC} = \text{SHA-256}(R_2 \oplus H_1) = \text{SHA-256}(0x5c5cf791bf72f9508c996418e086a5cb96e2399c92fde8a8dde7e1199fad265cf7323d3a) = 0x349808a3963b903444dc6cad5ffd5b11ff7a8edd42d244a605ffe0ee806250e2$$

Cet exemple illustre le processus HMAC avec des **valeurs concrètes** et les **constantes spécifiées**.

2.4 Troncation de la sortie du HMAC

2.4.1 Définition

Une fois le HMAC calculé, il est nécessaire de le tronquer pour obtenir un code de la longueur souhaitée. Pour ce faire, nous examinons les 4 bits de poids faibles du HMAC, cette valeur nous fournit l'offset à partir duquel extraire une portion de 4 bytes.

Ensuite, pour obtenir l'OTP, nous appliquons le modulo de 10^n pour obtenir un code à n chiffres. Avant de calculer le modulo, un masque est appliqué pour obtenir un entier non signé en big-endian de 31 bits. Ce processus permet de générer un code OTP de la longueur spécifiée à partir du HMAC calculé.

2.4.2 Exemple d'utilisation concret

Reprenons le résultat du HMAC de notre exemple, et tronquons le pour le transformer en OTP de 6 chiffres :

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Valeur	34	98	08	a3	96	3b	90	34	44	dc	6c	ad	5f	fd	5b	11
Index	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
Valeur	ff	7a	8e	dd	42	d2	44	a6	05	ff	e0	ee	80	62	50	e2

Ici les 4 bits de poids faible sont 0x2 ce qui nous donne l'index 2 à partir duquel on extrait les 4 bytes suivant : 0x08a3963b qui équivaut à 144938555 en base 10.

On calcule ensuite le modulo 6 de cette valeur : $144938555 \bmod 10^6 = 938\ 555$, ce qui correspond à notre OTP !

3 Prototype python de l'algorithme OTP

Afin de [mieux comprendre les concepts](#) expliqués dans la partie précédente et de mesurer les différents enjeux de sécurité discutés dans la première partie, nous avons décidé de réaliser une [implémentation de l'algorithme OTP dans le cadre du 2FA](#).

Pour réaliser ce prototype nous avons choisi d'utiliser [python](#) et la [librairie flask](#) pour le frontend. Dans un premier temps, La librairie [pyotp](#) qui implémente l'algorithme OTP a été utilisé pour accélérer le développement, puis nous avons décidé de créer [notre propre implémentation de l'algorithme OTP](#) selon les étapes expliqués dans la partie précédente.

Cette section vise donc à expliquer les **choix de conceptions** réalisés pour l'implémentation de l'algorithme et de montrer au lecteur comment [mettre en place une authentification à double facteurs](#) dans une application, dont les captures d'écran des différentes pages peuvent être consulté dans l'annexe-C.

3.1 Architecture et déploiement

Notre implémentation du protocole OTP se constitue d'une [application web](#) réalisée avec la librairie Flask de python. La Figure-18 représente l'arborescence des fichiers de ce projet. Le serveur '[gunicorn](#)' a été sélectionné afin de déployer l'application sur Heroku à l'adresse <https://dsc-securite-otp-c9eac3f8716a.herokuapp.com/>.

Le dossier '/app' contient le code source de l'application organisé comme suit :

app.py : Ce fichier est le point d'entrée de l'application et contient la [création et paramétrisation](#) de cette application, ainsi que les route des url '/'.
config.py Ces fichiers contiennent des [paramètres globaux](#) de l'application telle que la taille des codes OTP générés.

helpers.py Ce fichier contient les méthodes générales pour créer les pages d'[affichage d'information](#) aussi bien pour le serveur que pour le client, ce qui nous permet d'éviter la redondance de code.

myOTP.py Ce fichier contient l'[implémentation personnelle](#) de génération de code OTP , c'est à dire les fonctions [hmac](#), [troncation](#), et la [gestion du pas temporelle](#) de 30 secondes pour les TOTP .

share_seed.py Ce fichier contient les fonctions de [génération de Qrcode](#) afin de partager la seed à une application comme [freeOTP Authenticator-22](#) afin que l'utilisateur puisse générer ses codes sur son propre appareil.

/client et /server Ces répertoires contiennent la logique pour les url respectives '/client/' et '/server'.

Le projet contient aussi des fichiers spécifiques au déploiements listés ci-dessous :

.github/workflows/main.yml Ce fichier constitue notre [pipeline de déploiement](#) sur [héroku](#) lors du push dans la branche main de notre dépôt github.

Procfile Ce fichier est utilisé par Heroku pour [lancer l'application](#).

3.2 Utilisation de pyotp

La librairie **pyotp** est utilisé pour générer les seed avec la fonction **random_base32** qui selon la documentation effectue un tirage aléatoire dans une liste prédéfinie afin d'obtenir une seed.

```
user_secrets['user1'] = {  
    'TOTP': pyotp.random_base32(),  
    'HOTP': pyotp.random_base32(),  
    'HOTP_counter': 0  
}
```

FIGURE 10 – Génération d'une seed en base32 avec pyotp

3.3 Implémentation personnelle de l'algorithme OTP

Cette section présente une **capture d'écran des fonctions implémentées** pour la réalisation d'un algorithme générant des TOTP et HOTP en python afin de les répertoriées dans ce rapport et **rendre leurs consultation plus accessible**. Ainsi elle n'apportera pas plus d'explication par rapport aux explications précédentes et aux commentaires présent dans le code.

```
def generate_otp_qrcode(user, debug=False):
    otp_uri = {}

    for type in ['TOTP', 'HOTP']:
        seed = user_secrets[user][type]
        t = type.lower()
        otp_uri[type] = f'otpauth://{t}/{user}?secret={seed}&issuer={OTP_ISSUER}'
        print(otp_uri[type])

    # Générer le QR code
    for type, uri in otp_uri.items():
        qr = qrcode.QRCode(
            version=1,
            error_correction=qrcode.constants.ERROR_CORRECT_L,
            box_size=10,
            border=4,
        )

        qr.add_data(uri)
        qr.make(fit=True)

        img = qr.make_image(fill_color="black", back_color="white")

        # to be sent in the POST response
        img.save(f'{OTP_QRCODE_DIR}/{type}_qrcode.png')

    if debug:
        img.show()
```

FIGURE 11 – Création d'un uri et d'un QRcode pour le partage de la seed

```
def get_rounded_time_to_nearest_half_minute():
    """
    Return the time rounded to the nearest 30 seconds.
    """
    current_time = datetime.now()
    half_minute = timedelta(seconds=30)
    return int(int(time.time()) // 30)
```

FIGURE 12 – arrondissement du temps au multiple de 30s le plus bas

```

def generate_hmac(key, message, hash_algorithm='sha1') -> bytes:
    """
    Compute the HMAC value for the given key and message.

    Parameters:
    - key (bytes): The key to use for HMAC.
    - message (bytes): The message to authenticate.
    - hash_algorithm (str): The hash algorithm to use
      (default is SHA-1 to match the one used by freeOTP app).
    """
    block_size = 64 # Block size for SHA-1 and SHA-256

    # If the hash algorithm is SHA-512, adjust the block size
    if hash_algorithm == 'sha512':
        block_size = 128

    # If the key is longer than the block size, hash it
    if len(key) > block_size:
        key = hashlib.new(hash_algorithm, key).digest()

    # If the key is shorter than the block size, pad with zeros
    if len(key) < block_size:
        key += b'\x00' * (block_size - len(key))

    # XOR the key with the outer and inner pads
    o_key_pad = bytes(map(operator.xor, itertools.cycle(b'\x5c'), key))
    i_key_pad = bytes(map(operator.xor, itertools.cycle(b'\x36'), key))

    # Calculate the inner hash
    inner_hash = hashlib.new(hash_algorithm, i_key_pad + message).digest()

    # Calculate the outer hash
    outer_hash = hashlib.new(hash_algorithm, o_key_pad + inner_hash).digest()

    return outer_hash

```

FIGURE 13 – implémentation de HMAC part 1


```

def truncate_hmac_for_totp(hmac_value:bytes, code_length:int=6) -> int:
    """
    Truncate the HMAC value for TOTP protocol.

    Parameters:
    - hmac_value (bytes): The HMAC value to truncate.
    - code_length (int): The length of the TOTP code (default is 6).

    Returns:
    - int: The truncated TOTP code.
    """
    # Extract the last byte and the 4 least significant bits
    # i.e with do a binary AND with 00001111
    # which will 'remove' the lefts 4 bits
    offset = hmac_value[-1] & 0x0F

    # Extract a portion of 4 bytes from the HMAC value based on the offset
    truncated_bytes = hmac_value[offset:offset + 4]

    # Apply a mask to get an unsigned big-endian int of 31 bits
    # help : For the mask (7)16 = (0111)2, i.e remove the sign bit
    masked_value = int.from_bytes(truncated_bytes, byteorder='big') & 0x7FFFFFFF

    # Calculate the TOTP code using modulo 10^n
    totp_code = masked_value % (10 ** code_length)

    return totp_code

```

FIGURE 14 – implémentation de HMAC part 2

```

def generate_personal_hotp(user, key:str, counter:int, code_length:int=6, hash_algorithm:str='sha1') -> int:
    """
    Generate an OTP code for the given key and counter.

    Parameters:
    - key str : A str of a base32 encoded key. (generated with pyotp.random_base32())
    - counter (int): The counter to use for OTP.
    - code_length (int): The length of the OTP code (default is 6).
    - hash_algorithm (str): The hash algorithm to use (default is SHA-1).

    Returns:
    - int: The OTP code.
    """
    key = base64.b32decode(user_secrets[user]['HOTP'])
    message = counter.to_bytes(8, byteorder='big')
    hotp = generate_hmac(key, message)
    hotp = truncate_hmac_for_totp(hotp)

    return hotp

```

FIGURE 15 – implémentation de HOTP

```
def generate_personnal_totp(user, key:str, code_length:int=6, hash_algorithm:str='sha256') -> int:
    """
    Generate an OTP code for the given key and counter.

    Parameters:
    - key str : A str of a base32 encoded key. (g  n  r   avec pyotp.random_base32())
    - counter (int): The counter to use for OTP.
    - code_length (int): The length of the OTP code (default is 6).
    - hash_algorithm (str): The hash algorithm to use (default is SHA-1).

    Returns:
    - int: The OTP code.
    """
    current_time = get_rounded_time_to_nearest_half_minute()
    key = base64.b32decode(user_secrets[user]['TOTP'])
    message = current_time.to_bytes(8, byteorder='big')
    totp = generate_hmac(key, message)
    totp = truncate_hmac_for_totp(totp)

    return totp
```

FIGURE 16 – impl  mentation de TOTP

```
if otp_type == 'hotp':
    if VERSION_ALGO == 'perso' :
        hotp = generate_personnal_hotp(
            user,
            user_secrets[user]['HOTP'],
            user_secrets[user]['HOTP_counter']
        )
        valid = (hotp == int(otp_code))
    else :
        hotp = pyotp.HOTP(user_secrets[user]['HOTP'])
        valid = hotp.verify(otp_code, user_secrets[user]['HOTP_counter'])
    if valid:
        user_secrets[user]['HOTP_counter'] += 1

elif otp_type == 'totp':
    if VERSION_ALGO == 'perso' :
        print(f"seed = {user_secrets[user]['TOTP']}")
        totp = generate_personnal_totp(user, user_secrets[user]['TOTP'])
        valid = (totp == int(otp_code))
        print(f"totp : {totp} - otp_code : {otp_code} equals : {totp == otp_code}")
    else :
        totp = pyotp.TOTP(user_secrets[user]['TOTP'])
        valid = totp.verify(otp_code)
else:
    valid = False
```

FIGURE 17 – Serveur comparaison d'un OTP re  u

3.4 Mise en place du 2FA dans une application

Pour conclure, afin de [mettre en place un protocole 2FA](#) dans une application vous pouvez suivre les   tapes suivantes :

1. Trouvez une fonction de **g  n  ration de seed** al  atoire,

2. Utilisez une librairie de **génération de QRcodes** pour partager la seed grâce au format d'uri suivant :
`otpauth ://{totp, hotp}/{username}?secret={seed_base32_encoded}&issuer={issuer_name}` ;
et ainsi permettre à l'utilisateur de générer des OTP avec son application favorite.
3. utilisez une librairie de **générateur d'OTP** ou implémentez les fonctions ci-dessus
4. à la connexion, après que l'utilisateur a rentré son nom et mot de passe, affichez une pop-up pour **vérifier le code** OTP .
5. du côté du serveur, à la réception d'un code OTP générez un OTP en fonction de la seed de l'utilisateur et du temps ou du compteur de l'utilisateur afin de **vérifier la validité de l'OTP reçue** et **valider l'action** d'authentification.

4 Conclusion : failles de sécurité et bonnes pratiques

Pour conclure, OTP est largement adopté et **utilisé pour valider des actions** tels que l'authentification ou des transactions, mais **ne garantit pas une sécurité optimale** si on ne prend pas des précautions

- Faille sécurité partage de la seed
- Faille pour le partage via SMS/mail (fishing, interception de SMS)
- Code de 6 chiffres => brute force si pas de limite de tps fixé (généralement double le temps d'attente à chaque code erroné)
- Vols de recovery codes (code donnée à l'utilisateur après l'activation du 2FA pour le bypass avec un de ces codes en cas de perte de sa possession)

Il est donc conseillé de fixer un **paramètre d'étranglement** pour limiter le nombre de tentatives autorisées par utilisateur. Ou alors un **schéma de délais** pour qu'après chaque tentative de connexion, le serveur attend **A*T secondes** pour répondre (T souvent 5s).

On estime la sécurité de l'algorithme OTP mis en place avec la formule suivante :

$$Sec = s * v / 10^{Digit}$$

Avec :

Sec : probabilité de succès d'une attaque.

S : paramètre de la fenêtre d'anticipation (look-ahead window).

V : le nombre de tentatives autorisées.

Digit : le nombre de digit du code OTP.

Les deux systèmes OTP proposent des codes à usage unique, mais la **différence essentielle** réside dans le fait que, dans le système HOTP, un **HOTP** donné est **valable jusqu'à ce qu'il soit utilisé**, ou jusqu'à ce qu'un OTP ultérieur soit utilisé ce qui crée une vulnérabilité qui peut être utilisée par des attaquants.

Il existe d'autres **vulnérabilités** plus communes répertoriées dans le tableau suivant :

Nom de l'attaque	Déroulement de l'attaque	Défense
Phishing	Les attaquants trompent les utilisateurs pour obtenir leurs OTP via des e-mails, SMS ou des sites web malveillants.	Sensibilisation des utilisateurs, utilisation d'une authentification à deux facteurs (2FA) avec codes générés depuis une application pour renforcer la sécurité.
Attaque par interception de SMS (SWAP SMS)	Intercepter les OTP envoyés par SMS à l'aide de logiciels malveillants ou de dispositifs de surveillance.	Préférer des méthodes d'OTP ne reposant pas sur les SMS, tels que les applications d'authentification.
Brute Force	Les attaquants essaient différentes combinaisons pour deviner un OTP .	Imposer des politiques de complexité pour les OTP, tels que des codes plus longs et moins prédictibles. Définir un schéma de délais pour limiter le nombre de tentatives et doubler le temps d'attente à chaque erreur.
Attaque par rejeu	Intercepter un OTP valide et le réutiliser pour accéder au compte de l'utilisateur.	Utiliser des mécanismes de protection contre les attaques de rejeu, comme les horodatages.
Vol de clé secrète	Compromission de la clé secrète utilisée pour générer les OTP .	Stockage sécurisé des clés, rotation régulière des clés.
Utilisation d'applications tierces non sécurisées	Stockage ou transmission vulnérable des OTP par des applications tierces.	Utiliser des applications d'authentification reconnues et sécurisées.
Ingénierie sociale	Manipulation des utilisateurs pour divulguer leurs OTP en se faisant passer pour une autorité légitime.	Sensibilisation des utilisateurs, vérification stricte des demandes d'informations sensibles.

TABLE 1 – Tableau des attaques OTP et des défenses correspondantes

A Vocabulaire

Vocabulaire	Définition
2FA	double facteur d'authentification : protocole dans lequel un utilisateur doit prouver son identité en fournissant son mot de passe et en donnant un mot de passe généré/communiqué par un moyen que seul lui connaît.
AES	Advanced Encryption Standard, algorithme cryptographique symétrique
brute-force	attaque visant à tester toutes les combinaisons possibles afin de briser la sécuriter d'un système.
hexadécimal	notation en base 16 préfixée par 0x chaque chiffre peut prendre une valeur parmi [0-9AF].
HMAC	Mash Message Authentication Code : génère un code en utilisant une clé secrète, un message et unz fonction de hachage
IETF	: Internet Engineering Task Force, défini des standard d'Internet (non officiel), mais suivis par de nombreux dev.
iPad	constante de remplissage interne dans la fonction HMAC permettant d'homogénéiser la taille de la clé avec la taille de bloc de la fonction de hash configuré.
LFSR	Linear Feednack Shift Registration, algorithme de génération de séquence binaire pseudo-aléatoire.
oPad	constante de remplissage externe dans la fonction HMAC permettant d'homogénéiser la taille de la clé avec la taille de bloc de la fonction de hash configuré.
OTP	mot de passe à usage unique
seed	désigne une séquence servant d'origina à la génération d'une séquence pseudo-aléatoire.
XOR	opération binaire qui est vrai quand un seul des 2 bits est vrai.
Xorer	effectuer l'opération XOR entre 2 valeurs.

B Références

TOTP : Time-Based One-Time Password Algorithm	https://datatracker.ietf.org/doc/pdf/rfc6238
Research Article C-Lock : Local Network Resilient Port Knocking System Based on TOTP	https://downloads.hindawi.com/journals/wcmc/2022/9153868.pdf
Group Time-based One-time Passwords and its Application to Efficient Privacy-Preserving Proof of Location	https://eprint.iacr.org/2022/1280.pdf
HOTP : An HMAC-Based One-Time Password Algorithm	https://www.ietf.org/rfc/rfc4226.txt
Fuites de mots de passe	https://www.01net.com/actualites/246-milliards-d-identifiants-et-mots-de-passe-voles-circulent-sur-la-to.html
Étude sur la réutilisation de mots de passe	https://fr.statista.com/statistiques/967863/nombre-mots-de-passe-comptes-en-ligne-france/
Génération de secrets aléatoires	https://datatracker.ietf.org/doc/html/rfc4086
Échange de clés	https://eprints.qut.edu.au/31900/
Word list to make seed phrase	https://github.com/bitcoin/bips/blob/master/bip-0039/bip-0039-wordlists.md
Article intéressant sur Medium	https://medium.com/@nicola88/two-factor-authentication-with-totp-ccc5f828b6df
Code source d'une application d'authentification	https://freeotp.github.io/
Vidéo sur HOTP & TOTP	https://www.youtube.com/watch?v=XYVrnZK5MAU&list=PL3xlPfx1FbdbMafMn6sD3d3KEL3pGMpI7&index=2&t=7s
Vidéo sur TOTP	https://www.youtube.com/watch?v=VOYxF12K1vE&list=PL3xlPfx1FbdbMafMn6sD3d3KEL3pGMpI7&index=3&t=792s
Tutoriel vidéo sur pyotp	https://www.youtube.com/watch?v=o0XZZkI69E8&list=PL3xlPfx1FbdbMafMn6sD3d3KEL3pGMpI7&index=4&t=391s

Vidéo sur l'OTP envoyé par un fournisseur (SMS/email) n'est pas sécurisé

<https://www.youtube.com/watch?v=o0XZZkI69E8&list=PL3xlPfx1FbdbMafMn6sD3d3KEL3pGMpI7&index=4&t=391s>

Vidéo sur TOTP et la vie privée des données

<https://www.youtube.com/watch?v=ChKpf5HjcSY&list=PL3xlPfx1FbdbMafMn6sD3d3KEL3pGMpI7&index=6&t=1143s>

Étude vidéo sur Google Authenticator

<https://www.youtube.com/watch?v=2vP0gT5wGlA&list=PL3xlPfx1FbdbMafMn6sD3d3KEL3pGMpI7&index=7>

Vidéo sur les attaques sur 2FA et cet OTP

<https://www.youtube.com/watch?v=GexQHfT9fTE&list=PL3xlPfx1FbdbMafMn6sD3d3KEL3pGMpI7&index=8>

C Illustrations du prototype

<https://dsc-securite-otp-c9eac3f8716a.herokuapp.com/login>

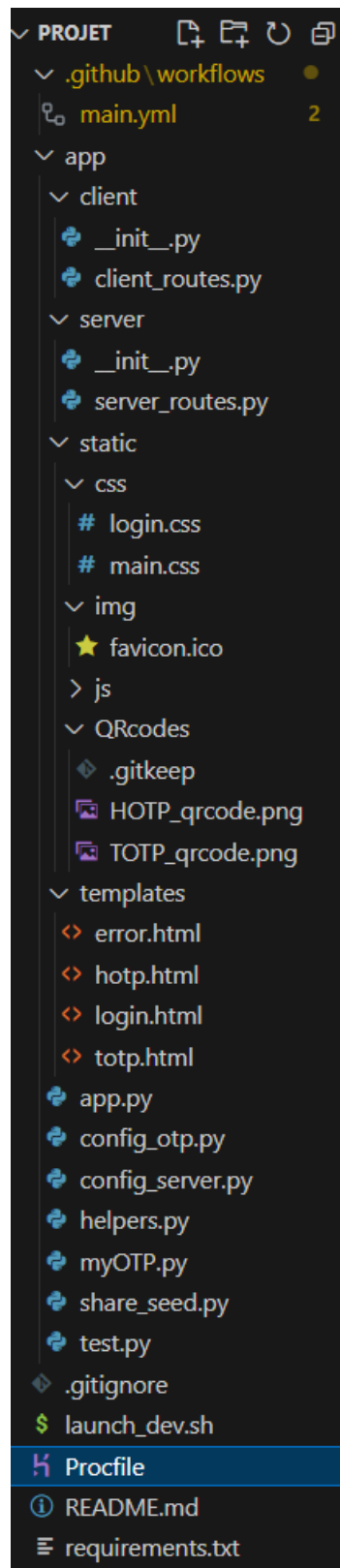


FIGURE 18 – architecture du projet Flask

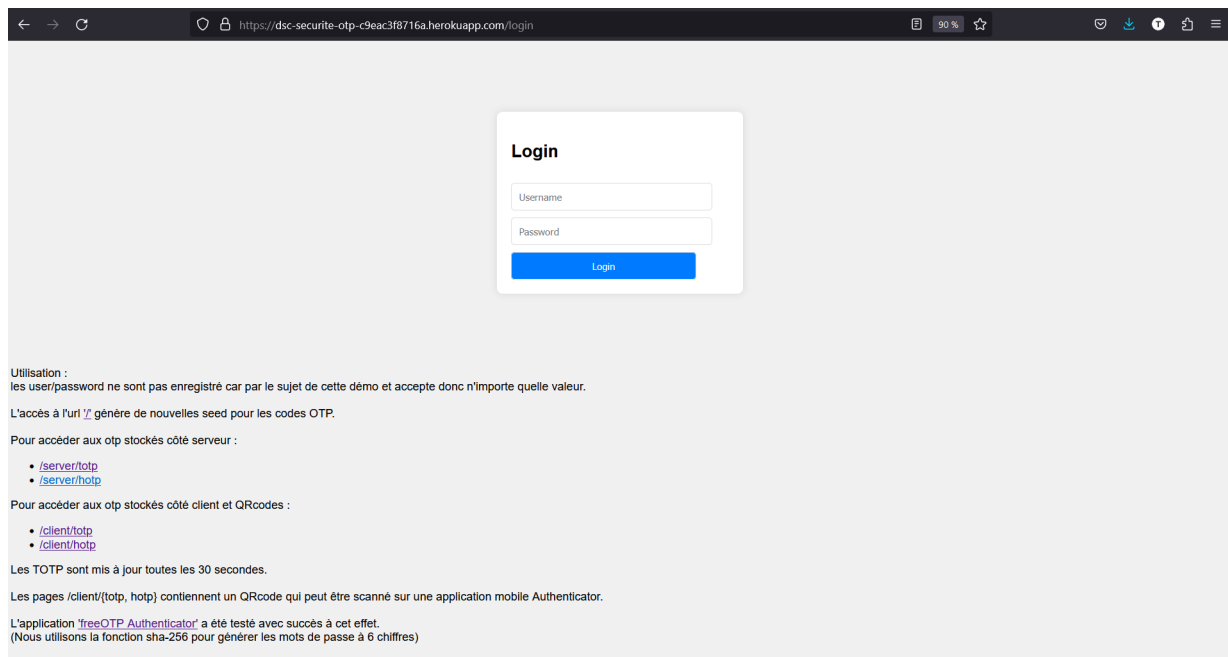


FIGURE 19 – Capture d'écran de la page principale

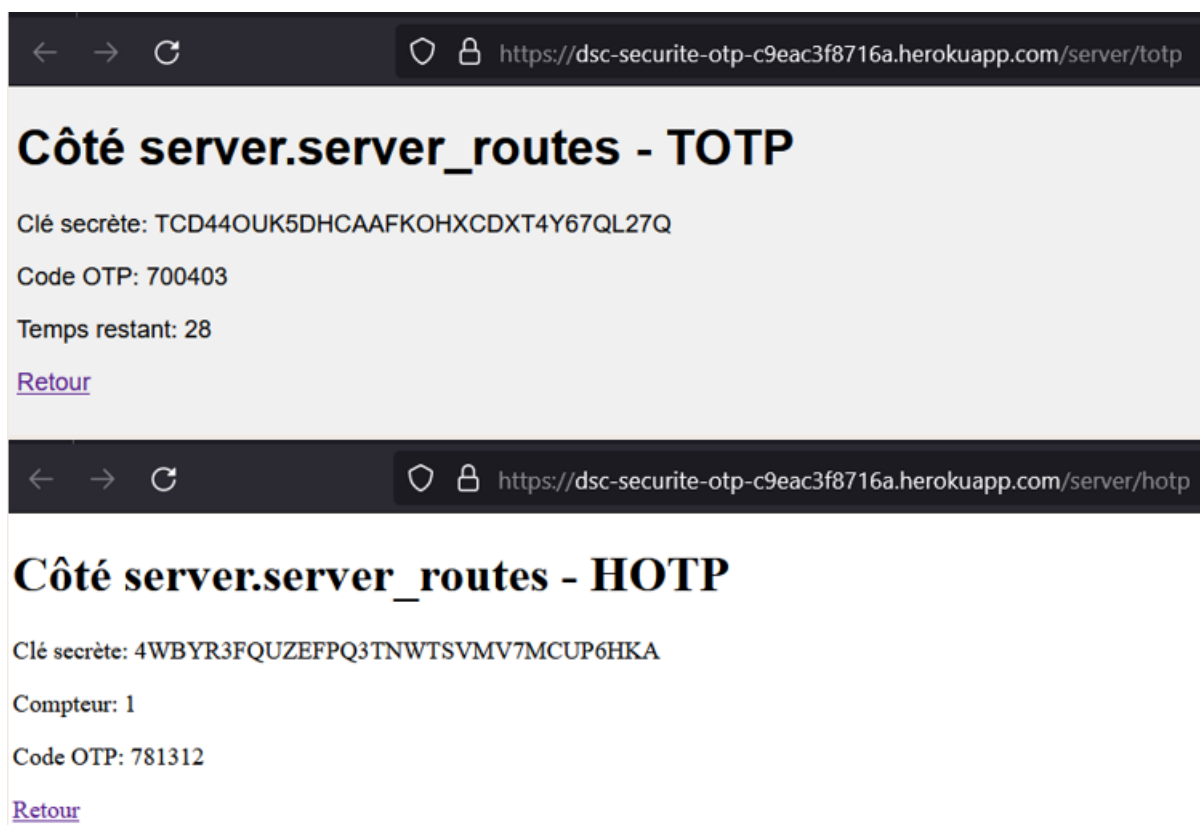


FIGURE 20 – Pages affichant les informations du serveur

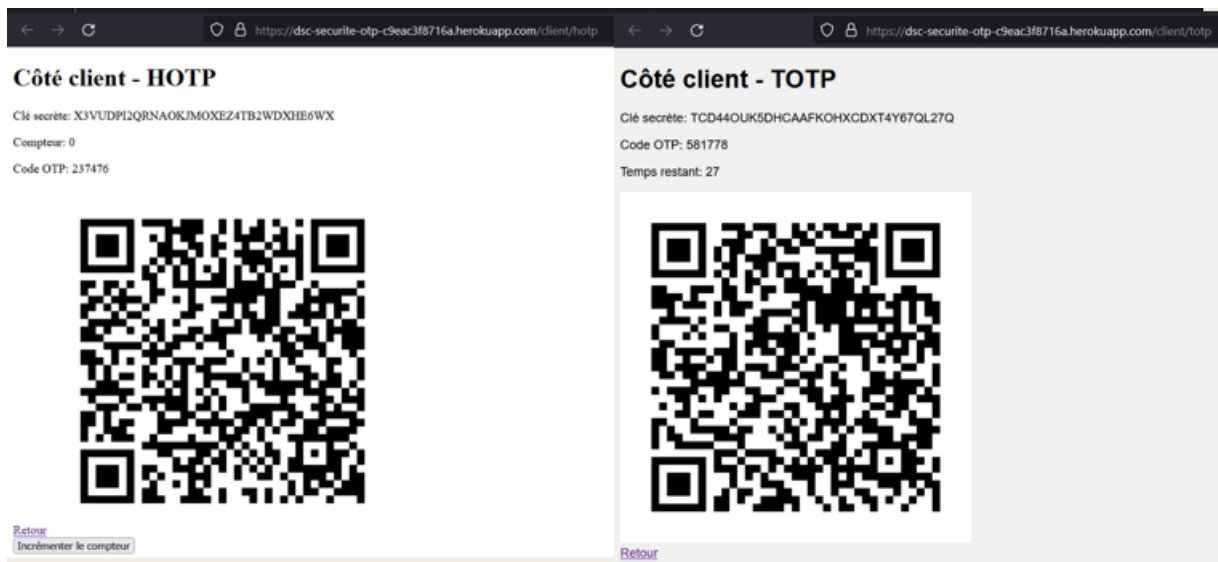


FIGURE 21 – Pages affichant les informations du client

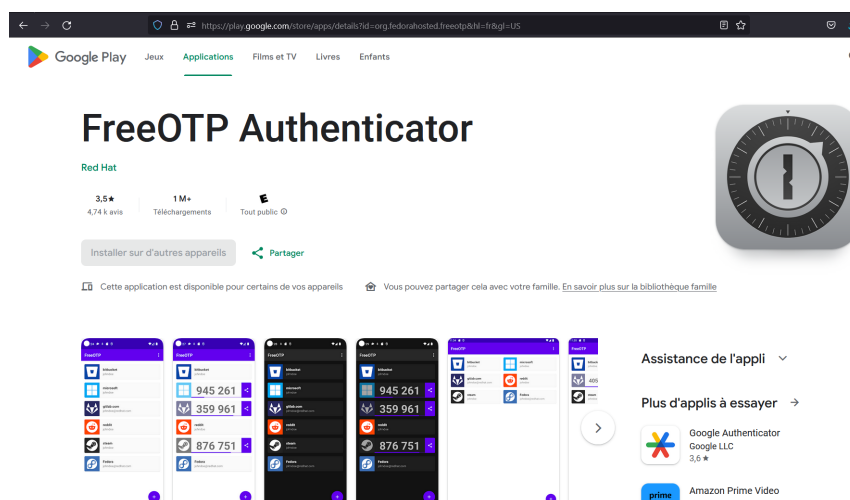


FIGURE 22 – FreeOTP Authenticator

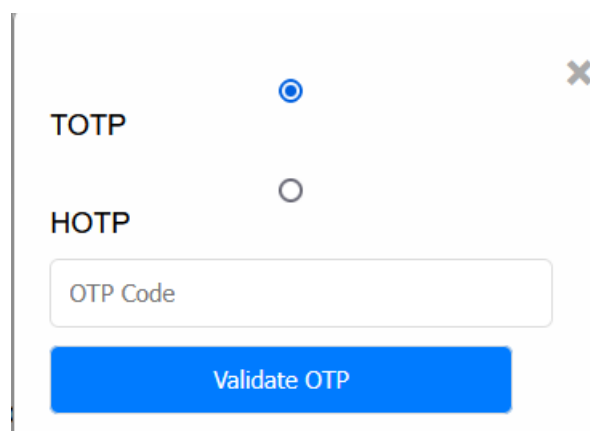


FIGURE 23 – pop-up de saisie du code

Login

**Connexion échouée avec OTP
invalid!**

Login

Login

**Connexion réussie avec OTP
valide!**

FIGURE 24 – Résultat de l'action