U2F2 : Prévenir la menace fantôme sur FIDO/U2F

Ryad Benadjila et Philippe Thierry
firstname.lastname@ssi.gouv.fr

ANSSI

Résumé. L'authentification à deux facteurs (2FA) devient un remplaçant de plus en plus répandu des méthodes d'authentification classiques basées principalement sur les mots de passe. Bien que ce second facteur puisse prendre plusieurs formes, l'alliance FIDO [3] a standardisé le protocole U2F [4] (Universal Second Factor) amenant un token dédié comme facteur. Le présent article discute de la sécurité de ces tokens au regard de leur environnement d'utilisation, des limitations des spécifications ainsi que de l'état de l'art des solutions apportées par l'open source et l'industrie. Un PoC implémentant des améliorations de sécurité, utiles dans des contextes sensibles, est détaillé. Il est fondé sur la plateforme open source et open hardware WooKey [26, 27] amenant de la défense en profondeur contre divers modèles d'attaquants.

1 Introduction

Le couple login/mot de passe, considéré comme une preuve de connaissance, fait partie des moyens d'authentification parmi les plus utilisés à l'heure actuelle [58]. Il souffre néanmoins d'inconvénients majeurs vis-à-vis de la sécurité. Les récents *leaks* de bases de données de divers services en ligne [34], compromettant ainsi les données de milliards d'utilisateurs, montrent à quel point ce facteur d'authentification trop simple est fragile. D'une part, il est intrinsèquement susceptible aux attaques par force brute, alors que la recherche exhaustive est de plus en plus simplifiée par les avancées des calculs parallélisés (par exemple avec les GPUs) ainsi que des logiciels open source tels John the Ripper [47] ou hashcat [9]. Ainsi, un mot de passe de 8 caractères alphanumériques peut être retrouvé en 10 jours sur un simple PC [10]. D'autre part, et au delà de cette fragilité inhérente, d'autres menaces viennent s'ajouter même contre les mots de passe considérés comme sûrs : le *social engineering*, le hameçonnage générique ou ciblé, les keyloggers sont autant de dangers supplémentaires.

Cet état de fait a poussé l'industrie à trouver des alternatives pour une *authentification forte* utilisant d'autres facteurs. En plus de la preuve de connaissance, il s'agit de demander à l'utilisateur légitime qui s'authentifie

de produire une preuve de possession (via des objets physiques dédiés), une preuve d'identité (via de la biométrie), etc. Plusieurs solutions utilisant un second facteur, en plus du login/mot de passe, ont alors émergé et coexistent à l'heure actuelle. Les logiciels Google Authenticator [8] et FreeOTP [7] génèrent des mots de passe à usage unique (OTP, pour One Time Password) comme second facteur. RSA SecureID [14] est un périphérique dédié avec un écran embarqué et générant des OTP. Bien que les OTP soient un premier pas vers une meilleure sécurisation de l'authentification, ils ne constituent pas une solution idéale : ils sont susceptibles au hameçonnage [52], et dépendent fortement de la sécurisation de leur canal de délivrance (par exemple, les vulnérabilités dans le protocole SS7 [57] ou les services de détournement des messages [59] écornent la fiabilité des SMS).

D'autres preuves de possession existent et sont déployées et utilisées depuis des années par l'industrie : les cartes à puce. Elles utilisent une technologie et des composants sécurisés à la robustesse éprouvée contre divers attaquants distants et locaux, notamment les attaques matérielles (par canaux auxiliaires et attaques en fautes). Leur déploiement auprès du grand public s'est néanmoins heurté à plusieurs obstacles : une intégration complexe et des *middlewares* propriétaires et peu portables les ont cantonnées au monde professionnel.

Afin d'amener une solution à la fois flexible et sécurisée pour l'authentification deux facteurs (2FA), le consortium FIDO [3] fut fondé en 2013. De ses réflexions émergea le standard FIDO 1.2 (U2F) en 2017 [4], puis son successeur FIDO 2.0 (CTAP) en 2019 [1], et la version 2.1 qui est en cours de publication. Ce standard met en avant l'utilisation d'un périphérique dédié nommé *token*, se branchant en USB, NFC ou Bluetooth, et permettant d'assurer un second facteur d'authentification auprès des services compatibles. Afin d'assurer une interopérabilité complète, les spécifications FIDO décrivent précisément les échanges à établir lors de phases d'enregistrement et d'authentification aux services au travers des navigateurs web. Force est de constater que le pari de FIDO est gagné vu l'adoption en hausse auprès du grand public [32, 53]. Aujourd'hui, FIDO est même de plus en plus vu comme un facteur d'authentification unique, prêt à complètement remplacer le mot de passe [51], imposant de ce fait des contraintes de sécurité d'autant plus fortes sur le token. La sécurité du standard FIDO a été étudiée et les risques et fonctions de sécurité y répondant spécifiées [23]. Néanmoins, les suppositions sur le *Client* (i.e. le PC sur lequel le token se connecte) sont particulièrement strictes et peu réalistes dans certains contextes d'utilisation, par exemple

en cas de nomadisme, l'équipement terminal devant être impérativement de confiance [23] : « *SA4 : The computing environment on the FIDO user device and the end applications involved in a FIDO operation act as trustworthy agents of the user.* » Le modèle de sécurité sur lequel s'appuie FIDO est alors fortement affaibli, comme le précise le consortium lui-même un peu plus loin dans le document : « *FIDO can also provide only limited protections when a user chooses to deliberately violate [SA-4], e.g. by roaming a USB authenticator to an untrusted system like a kiosk, or by granting permissions to access all authentication keys to a malicious app in a mobile environment.* ».

De plus, malgré une cryptographie solide et des preuves sur le protocole en lui-même [24, 36, 49], les spécifications FIDO laissent à la discrétion de l'implémentation divers éléments pouvant fortement impacter la sécurité. Bien que la sécurité côté services et navigateurs soit primordiale et que certaines dérives ont ouvert des voies d'exploitation [29, 50], nous nous focalisons dans le présent article exclusivement sur la sécurité du token et ses possibles améliorations en réponse aux limitations du modèle et de l'état de l'art.

Nous présentons tout d'abord en quoi l'état de l'art des tokens existant souffre de diverses limitations lorsqu'il s'agit de les utiliser dans des contextes sensibles. D'une part, beaucoup de tokens ne prennent pas en compte la sécurité logicielle dans les firmwares déployés : pas de séparation de privilèges, des mises à jour peu ou mal protégées, etc. La sécurité matérielle contre les attaques locales (canaux auxiliaires, fautes) n'est en général pas prise en compte. L'aspect propriétaire et fermé de certains tokens, malgré leur possible robustesse et leur architecture interne pertinente, ne permet pas d'en auditer le réel niveau de sécurité. De plus, et c'est une chose primordiale, aucune authentification locale de l'utilisateur n'est utilisée : l'utilisateur ayant un token peut manifester son consentement lors d'une authentification simplement via un appui sur un bouton. Dans des scénarios de vol, vol avec remise, attaques logicielles prenant la main sur le firmware, il est possible de cloner le token [44], parfois sans même que l'utilisateur n'en soit conscient (si les services implémentent mal les spécifications).

Enfin, le standard FIDO est particulièrement sensible aux attaques exploitant une désinformation de l'utilisateur sur le token. L'absence de vérification possible de la source de l'authentification au niveau même de ce matériel est source de confusion : en effet, lorsque le bouton du token s'illumine, l'utilisateur ne se pose en général pas la question du service initiant la requête. Si un *malware* exploite un moment où l'utilisateur utilise

un service légitime, l'attaquant peut enregistrer un service malveillant ou s'authentifier auprès d'autres services à son insu (en ayant volé le mot de passe au préalable, via un *keylogger* ou autre *phishing*). Le seul effet visible pour l'utilisateur sera un échec de son opération légitime qu'il mettra vraisemblablement sur le compte d'un mauvais appui.

Partant de ce constat, nous présentons nos diverses améliorations pour amener un PoC de token FIDO U2F durci à des fins d'usage dans des contextes sensibles (nomadisme, authentification à des services contenant de la propriété intellectuelle d'entreprise, etc.). Nous bâtissons ce token sur la base du SDK de la plateforme WooKey [28], héritant ainsi de défense en profondeur éprouvée [17] ainsi que de modules d'authentification locale déjà présents. Nous étendons cela via le développement de bibliothèques spécifiques à FIDO, et nous discutons en détail nos choix concernant la cryptographie ainsi que la séparation en tâches au sein du token. Nous amenons aussi des éléments d'usage du token intéressants, possibles grâce à l'écran tactile de WooKey, et non couverts par les spécifications FIDO : la notification à l'utilisateur des services qui demandent une authentification, pour contrer les attaques par confusion, ainsi que la fourniture d'un mécanisme de désenregistrement volontaire des services au niveau du token.[1]

2 FIDO/U2F : le *reader's digest*

2.1 Principes généraux et historique

FIDO (Fast Identity Online) est un consortium fondé en 2013 dans le but d'apporter un standard d'authentification basé sur un périphérique matériel afin d'amener un second facteur d'authentification pour les services en lignes. Le consortium est fondé par des entités comme PayPal, Lenovo, ou Infineon, et est vite rejoint par Google, NXP et Yubico, première société à se lancer dans la fabrication de périphériques matériels implémentant le standard.

Le standard est conçu pour être interopérable, anonyme (sans mécanisme d'enrôlement) avec un fort accent sur l'expérience utilisateur. Il se base sur un système trois-tiers présenté sur la Figure 1 :

1. le **Service** ou *Relying Party* (*RP*) : il s'agit du service web sur lequel l'utilisateur souhaite s'authentifier (Gmail, Facebook ou GitHub par exemple) ;

1. Suite à une désinscription d'un service par exemple.

2. le **Browser** : le navigateur web du poste client (Firefox, Chrome, etc.). Cette entité permettra de communiquer avec les éléments matériels. Par extension, nous incluons également sous cette appellation l'ensemble du poste client (machine, système d'exploitation, périphériques et interfaces) ;
3. le **token**, en charge des opérations cryptographiques permettant d'authentifier le porteur. Le protocole U2F prévoit l'utilisation de token communiquant par les interfaces USB, NFC ou Bluetooth.

La communication entre service et navigateur est transportée sur du protocole HTTPS, la communication entre navigateur et token est formalisée successivement sous deux formes. En 2017, la version 1.2 du standard [4] est publiée, basée sur une communication utilisant la classe HID (Human Interface Device) de type U2F [5], encapsulant des APDU (Application Protocol Data Unit) [6] utilisés dans le domaine des cartes à puce. En 2019, la version 2.0 du standard substitue le standard U2F par CTAP (Client to Authenticator Protocol) [1], plus riche, et l'usage des APDUs par le format CBOR (Concise Binary Object Representation) [30] basé sur le formalisme JSON [31], plus élaboré et standardisé mais posant un problème de complexité des parseurs [15, 16].

Dans le cadre de nos travaux, nous nous intéressons à l'implémentation du périphérique (token) pour du FIDO U2F 1.2. Celui-ci est un élément transportable, perdable, et sur lequel repose une grande partie de la sécurité du modèle FIDO. Nous nous sommes focalisés sur l'USB comme mode de communication avec le navigateur : notre démonstrateur s'appuie sur une interface USB HID [5] sur laquelle notre preuve de concept est la plus aboutie. Néanmoins, l'ensemble des principes architecturaux que nous allons présenter sont applicables aux deux autres modes de communication et n'ont pas de dépendances.

Les spécifications du standard donnent l'ensemble des interactions entre chacune des entités, mais laissent libres les implémentations internes. Ainsi, les contraintes de sécurité sur le token (stockage, mécanismes cryptographiques n'impactant pas les interfaces) ne sont pas spécifiées ou imposées par le standard. Cela amène de possibles ambigüités, mais aussi de la flexibilité notamment concernant l'amélioration de la sécurité sans violer le standard comme nous allons le voir.

Du point de vue des échanges entre le navigateur et le périphérique, le protocole de communication FIDO est simplifié à l'extrême. Celui-ci est basé sur deux étapes successives :

— Une étape d'enregistrement du token auprès du service, nommée `REGISTER`. Celle-ci permet de créer une empreinte unique (via une

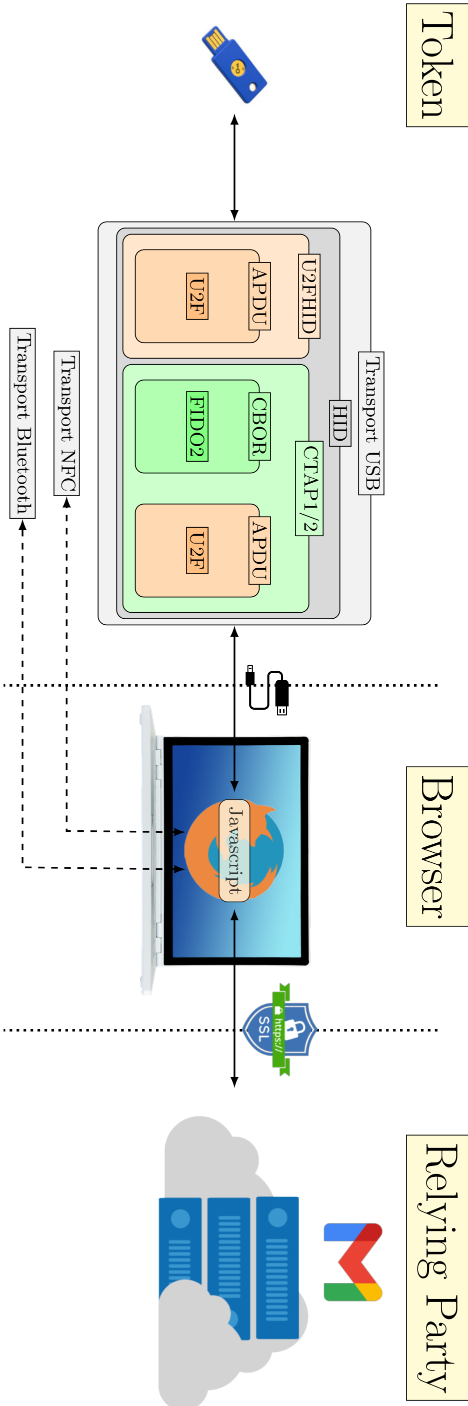

Fig. 1. Architecture trois-tiers de FIDO/U2F (1.2) et FIDO2

génération de clé cryptographique ECDSA) que seul le périphérique est capable de reproduire dans le futur, et qui est conservée par le service. Cette empreinte est nommée *Key Handle*.
— Une étape d'authentification du token auprès du service (via une signature ECDSA utilisant la clé générée lors de l'enregistrement), nommée `AUTHENTICATE`. Celle-ci permet de valider que c'est bien le token qui a servi à la phase d'enregistrement qui est utilisé : le service envoie le *Key Handle* permettant au token de dériver la clé ECDSA acquittée à l'enregistrement.

Il n'existe pas de notion de « désenregistrement ». La perte du token doit donc impliquer, de la part de l'utilisateur, l'usage d'un mécanisme tiers d'authentification auprès du service (e-mail de récupération, OTP, SMS, etc.) puis la suppression du token de la méthode d'authentification.[2] FIDO fournit un mécanisme de signature via certificats X.509 permettant d'authentifier le token et possiblement une famille de tokens (hiérarchie de certificats). Cette signature n'est néanmoins en général pas vérifiée et les mécanismes de révocation permettant de rendre un token inactif ne sont pas exploités par les services alors que le protocole le permettrait.

3 U2F2 : défense en profondeur

3.1 Modèle d'attaque et protections proposées

Plusieurs modèles d'attaques peuvent être considérés sur chaque maillon de l'architecture trois-tiers de FIDO/U2F : contre le browser, contre les services, contre le token, contre les liens entre ces éléments.

Dans le cadre de cet article, nous nous concentrons sur la *sécurisation du token*, et plus particulièrement contre les menaces d'attaques logicielles et d'attaques physiques (pour de l'utilisation illégitime suite à un vol, clonage ou piégeage suite à un vol avec remise). En effet, la plupart des solutions existantes open source et même à sources fermées [11, 37, 54] n'apportent pas vraiment de solution d'authentification locale de l'utilisateur : le bouton de *user presence*, généralement un simple bouton sur lequel appuyer sans authentification préalable de l'utilisateur, n'est pas protégé comme le montre la Figure 2 sur un modèle de Yubikey largement déployé. Seules exceptions à notre connaissance : les clés Yubikey intégrant de la biométrie [39] mais dont la sécurité n'est pas éprouvée, les clés

2. La capacité à toujours pouvoir récupérer une authentification par une méthode tierce potentiellement plus faible (par exemple SMS) peut casser le modèle de sécurité de FIDO.

Ledger Nano [18] qui sont en sources fermées, les clé Trezor [60] et les OnlyKeys [19] qui présentent un PIN pad mais qui sont limitées par ailleurs en terme de sécurité logicielle et matérielle du fait de leur *design*. Dans des contextes sensibles où l'assurance de la légitimité d'un utilisateur est critique, cette authentification locale forte de l'utilisateur manque cruellement.

Fig. 2. Yubikey : bouton poussoir pour le *user presence*

Par ailleurs, l'état des lieux des solutions FIDO/U2F existantes montre que celles-ci manquent de *défense en profondeur* sur le token, à la fois en ce qui concerne la *sécurité logicielle* mais aussi sur le sujet de la *sécurité matérielle* :

— Du côté logiciel, beaucoup de solutions open source ne mettent pas en œuvre de cloisonnement (au sens OS et privilèges) [37, 54]. La moindre vulnérabilité exploitable dans un des éléments FIDO (USB par exemple) permet à un attaquant de prendre la main au niveau de privilège le plus élevé. Cela est d'autant plus critique lorsque le protocole impose du *parsing* non trivial de paquets. Concernant les solutions industrielles privées (Yubikeys et autres), il n'est pas simple de statuer sur leur niveau de sécurité et le cloisonnement qu'elles mettent en œuvre. Des initiatives intéressantes comme OpenSK [38] amènent via Rust et TockOS [13, 43] des paradigmes de sécurité logicielle, mais restent limitées d'un point de vue sécurité matérielle (voir ci-dessous), et ne prennent pas en compte les problématiques de sécurisation des mises à jour de firmware.

— Concernant la sécurité matérielle, à savoir la protection contre les attaques par canaux auxiliaires (SCA) ou les attaques par injection de fautes (FA), les solutions open source offrent des protections très limitées voire inexistantes [33, 46]. Les solutions commerciales implémentent normalement des contre-mesures en

utilisant notamment des composants sécurisés (issus du monde des cartes à puce), ce qui ne les empêche pas de se faire attaquer [44] lorsque ce composant est la seule source de confiance ou que celui-ci présente des faiblesses [45, 62] : cela est d'autant plus gênant lorsqu'aucune mise à jour n'est possible (ce qui est souvent le cas pour ces composants), amenant un coûteux rappel et remplacement des tokens. Les attaques matérielles simples ainsi que les attaques hybrides deviennent de plus en plus accessibles aux attaquants en utilisant du matériel abordable [17] : cela augmente la menace sur les tokens en cas de vol et vol avec remise.

Une des rares solutions alliant à la fois sécurité locale et robustesse matérielle est le Ledger Nano (S et X). Cette solution souffre néanmoins de deux limitations : elle est en grande partie en sources fermées (donc difficilement auditable), et l'élément sécurisé utilisé est sur le PCB enlevant l'aspect « troisième » facteur que nous décrivons ci-dessous.

Nous considérons également dans cet article les solutions à l'attaque par confusion de l'utilisateur. Cette attaque n'est pas considérée dans le modèle FIDO [23] du fait de ses hypothèses et il n'existe, à notre connaissance, aucune solution permettant d'y répondre complètement. Du côté open source, le Trezor [60] implémente une reconnaissance des services qui s'enregistrent ou s'authentifient avec acquittement de l'utilisateur sur l'écran. Du côté propriétaire, seul le Ledger Nano [18] propose cela. Néanmoins, aucun des deux ne semble offrir la possibilité de désenregistrement des services qui nous semble importante pour pallier les lacunes du standard.

Forts de ce constat, nous avons essayé d'amener des réponses à ces attaques logicielles et matérielles en utilisant la plateforme open source et open hardware WooKey [28]. Nous utilisons notamment directement : le module d'authentification locale mis en place via l'écran tactile de saisie du PIN, la sécurisation amenée par le composant sécurisé de la carte à puce d'authentification, la défense en profondeur logicielle (microkernel, séparation en tâches, etc.).

Le travail d'innovation que nous allons présenter a consisté en le développement et l'intégration à cette plateforme des modules FIDO/U2F nécessaires en ayant réfléchi à une architecture logicielle et cryptographique pertinente.

Les protections apportées renforcent fortement la sécurisation du token contre les attaques logicielles venant du PC hôte sur lequel il est branché, contre le vol et le vol avec remise et limitent fortement le piégeage. La carte à puce *externe* constitue un troisième facteur fort d'authentification,

l'attaquant ne pouvant rien faire avec le WooKey principal ou la carte à puce seuls (grâce à une ségrégation cryptographique détaillée dans la section 4). De plus, si le composant de la carte possède une faiblesse comme pour [44,45,62], il est possible de le remplacer aisément via la compatibilité des applets Javacard de WooKey avec toutes les cartes compatibles.

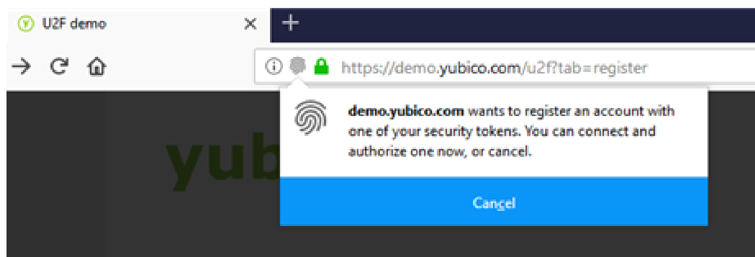

Fig. 3. Infobulle FIDO/U2F sur Firefox

Grâce à l'écran tactile embarqué sur WooKey, nous luttons contre les attaques par confusion en amenant l'information qu'une demande est faite à l'utilisateur jusqu'au périphérique (ce type d'élément existe au niveau du browser comme montré sur la Figure 3 mais pas au niveau des tokens). De plus, il est possible d'avoir une *mémorisation* des services enregistrés pour les *révoquer* (non prévu par le protocole lui-même). Grâce à l'écran tactile et au stockage local sécurisé SD de WooKey, nous permettons à l'utilisateur de révoquer lui-même un service qu'il aurait enregistré, lui (re)donnant le contrôle sur ces éléments.

Enfin, nous pouvons aussi implémenter différentes stratégies d'attestation plus ou moins strictes en fonction de la politique de sécurité envisagée : le standard FIDO prévoit un champ spécifique de *user presence* qui rend l'appui sur le bouton optionnel (décidé par le browser) lors des authentifications. Dans certains contextes ou pour certains services critiques (configurables par l'utilisateur ou un officier de sécurité), il est souhaitable de ne pas tenir compte de ce champ et de systématiquement s'assurer du consentement de l'utilisateur.

3.2 Comparaison à l'état de l'art des tokens FIDO/U2F

Lorsque nous mettons bout à bout les diverses propriétés de sécurité qui nous semblent nécessaires pour assurer une protection efficace de l'utilisateur FIDO U2F dans un contexte de nomadisme pour s'authentifier

à des services sensibles, peu de solutions cochent toutes les cases comme le montre la Table 1. Sans être exhaustifs, nous résumons ci-après les points manquants notables de chaque solution :

— **SoloKeys** [54] : ces clés open source n'offrent pour l'instant pas de protection logicielle solide (bien qu'une future version prévoie une refonte en Rust du firmware). Elles n'utilisent pas de composant sécurisé, et sont dès lors fortement susceptibles aux attaques matérielles. Aucun élément permettant de palier les ambigüités ou lacunes de FIDO (anti-confusion utilisateur, authentification locale forte, désenregistrement de services) n'est apporté.

— **OpenSK** [38] : hormis la sécurité logicielle apportée par Rust, pas ou peu d'effort n'est amené pour l'instant côté sécurité matérielle. L'aspect mise à jour (lié à TockOS et au cycle de vie de ses applications) est laissé de côté.

— **Nitrokey3** [20] : l'aspect open source est mis en avant par les concepteurs, mais pour l'instant seule l'utilisation de Trussed [21] pour la cryptographie est confirmée. En l'absence de plus d'information, il est difficile d'évaluer la sécurité de ces clés. Par ailleurs, le composant sécurisé utilisé est un SE050 de NXP impossible à mettre à jour et impossible à changer car soudé sur le PCB.

— **Ledger Nano** [18] : les Ledger Nano (X et S) ont le gros avantage d'amener de l'authentification forte locale sur le token, ainsi que de l'anti-confusion sur l'écran du *device*. Cependant, ces clés souffrent d'un composant sécurisé soudé sur PCB, et du fait que beaucoup d'éléments sont en sources fermées empêchant d'évaluer une sécurité qui semble néanmoins avancée d'après les communiqués publics [42].

— **Trezor** [60] : les Trezor sont des crypto-wallets amenant de l'authentification forte sur le token (du moins dans leur version Trezor T). Le firmware de ces tokens est open source. Ces *devices* souffrent néanmoins de deux problèmes majeurs d'un point de vue sécurité : la protection logicielle du firmware reste limitée (usage de micropython pour cloisonner les applications), et surtout l'absence de composant sécurisé pour protéger les secrets a amené des attaques physiques très peu chères (environ 100 $ d'équipement) mais critiques [35,40]. Il faut cependant noter la présence utile, pour FIDO, d'anti-confusion sur l'écran tactile.

— **Yubikeys** [11] : les Yubikeys, pionnières du FIDO, contiennent un composant sécurisé et un minimum de protections qu'il est difficile d'évaluer sans accès aux sources ou spécifications détaillées. Parmi les grosses lacunes, il n'y a pas de mise à jour du firmware

prévue, et l'authentification locale « forte » présente sur les versions biométriques [39] est difficilement évaluable face à une solution utilisant un PIN à essais limités. Ces clés ne règlent pas les autres lacunes de FIDO.

— **OnlyKeys** [19] : ces clés open source ont l'avantage d'amener un PIN pad physique sur le périphérique pour une authentification locale forte. Elles souffrent par contre à côté de cela de plusieurs limitations en termes de sécurité logicielle et matérielle.

	SoloKeys [54]	OpenSK [38]	Nitrokey3 [20]	Ledger Nano [18]	Yubikeys [11, 39]	OnlyKeys [19]	Trezor [60]	U2F2
Open Source	x	x	~	~		x	x	x
Firmware cloisonné		x	~	x	?		~	x
Mise à jour sécurisée du firmware	x		x	x		?	~	x
Résistance SCA et fautes			x	x	x			x
Mise à jour logicielle du composant sécurisé				x				x
Changement de composant sécurisé								x
Authentification locale forte				x	~	x	x	x
Anti-confusion utilisateur				x			x	x
Désenregistrement de services								x

Tableau 1. Comparaison de U2F2 à l'état de l'art des tokens FIDO/U2F

Notons que certaines des solutions précédentes (comme les Nitrokey3) offrent une authentification par PIN depuis le PC via une application dédiée. Nous ne considérons pas cela comme une authentification forte car dans le modèle d'attaque par *malware*, *keylogger* et autre hameçonnage, cette solution est très fragile. Seule une saisie physique locale sur le périphérique lui-même assure une bonne sécurité.

Ainsi, nous essayons d'amener avec U2F2 un PoC répondant à toutes les contre-mesures qui nous semblent utiles à un utilisateur soucieux de la sécurité de son token dans un environnement hostile. Concernant les limitations de la solution que nous amenons dans cette soumission, il y a la compatibilité avec le seul standard FIDO U2F (1.2), alors que la plupart des solutions précédemment décrites supportent FIDO 2 ainsi que d'autres standards connexes (OTP, KeePass, PGP, etc.). La compatibilité

FIDO 2 fait partie des travaux à venir à court et moyen termes, de même que l'extension aux autres standards classiques pour ce genre de clés (à moyen et long termes), en gardant la même philosophie de défense en profondeur qui caractérise notre approche de la conception architecturale logicielle et matérielle.

3.3 Coût estimé du token U2F2

Lorsqu'il s'agit de solutions grand public implémentant FIDO, la question du prix est importante. Il est évidemment primordial de garder une enveloppe de prix raisonnable sous peine de rendre la solution inaccessible, ou seulement dans des contextes d'usages professionnels avancés.

La plateforme WooKey étant un prototype, elle est évidemment susceptible aux facteurs d'échelle lorsque l'industrialisation est considérée. D'après [27], la fabrication de 1000 PCBs amène un prix de 50€ à peu près par carte nue, à ajouter aux environs 5€ par carte pour 1000 cartes à puce, plus le coût de la microSD et du boîtier. L'enveloppe d'industrialisation est donc estimée aux environs de 70€, ce qui est dans la fourchette haute des produits FIDO de la gamme Yubikey par exemple.[3]

Des optimisations de l'architecture matérielle sont aussi possibles pour diminuer ce prix. Beaucoup de modules du SDK étant également portables et non adhérents au microcontrôleur STM32F4 du prototype, il est aussi envisageable à moyen terme de changer d'architecture matérielle en gardant les mêmes principes d'architecture logicielle.

4 U2F2 : détails et résultats

Dans la présente section, nous détaillons les innovations amenées sur la plateforme WooKey. Ainsi, nous laissons le lecteur se référer aux articles [26, 27] d'origine décrivant le détail du fonctionnement de cette plateforme. Nous nous concentrons principalement sur les axes qui différencient U2F2, à savoir : la cryptographie dédiée à U2F, l'utilisation de la défense en profondeur déjà présente et la séparation en tâches au sein du SDK WooKey adaptée à FIDO, le développement du système anti-confusion et de « désenregistrement » des services, et enfin les indicateurs de bon fonctionnement et de conformité du token.

3. Avec la présence en plus d'un écran tactile couleur permettant diverses cinématiques, et une versatilité de la plateforme pour d'autres usages (clé USB chiffrante, etc.).

4.1 Sécurités directement héritées de WooKey

Authentification au démarrage : Le token U2F2 est protégé par le mécanisme d'authentification au démarrage de WooKey basé sur le couplage de l'équipement et de la carte à puce Javacard, impliquant un déverrouillage en deux étapes par l'utilisateur par l'entrée de deux codes PINs : *PetPIN* et *UserPIN*. L'utilisateur saisit ses PINs sur l'écran tactile, ceux-ci étant transmis à la carte à puce avec blocage en cas d'essais maximums infructueux (et destruction du matériel cryptographique dans ce cas pour plus de sécurité).

Séparation en tâches : La plateforme offre le micronoyau EwoK ainsi que la séparation en différentes tâches cloisonnées et discutant via des IPC (Inter-Process Communication). Nous décrivons cette séparation plus en détail en section 4.3. La chaîne de compilation est de plus durcie avec l'utilisation de *stack canaries* et autres vérifications de pointeurs de fonctions (*handlers*). L'usage des périphériques est strictement limité à son minimum, et la déclaration de ceux-ci ne peut se faire que dans la phase de démarrage et en accord avec la matrice des droits du SDK : la prise de contrôle d'une tâche post-démarrage ne permet pas d'utiliser d'autres périphériques que ceux déclarés.

Enfin, notons que du côté de la pile USB l'interface n'est exposée au PC hôte qu'après l'authentification réussie de l'utilisateur via ses deux PINs, limitant ainsi la surface d'attaque. De plus, le cœur de cette pile USB et la classe HID utilisée par U2F ont été analysés par le biais de méthodes formelles dans un article connexe [25], apportant certaines garanties contre les RTEs (Run Time Errors).

Mise à jour du Token : Le token U2F2 peut être mis à jour via un second mode de démarrage offrant une interface USB DFU (Device Firmware Upgrade). Ce second mode de démarrage est en tout point similaire et utilise les mêmes modules logiciels (bibliothèques, tâches, drivers) que celui offert par le disque USB sécurisé WooKey. Il impose donc l'usage d'une carte à puce Javacard dédiée permettant de différencier les rôles du porteur et du gestionnaire de mise à jour, et permettant de protéger le token FIDO contre toute tentative de mise à jour non authentifiée. L'ensemble des mécanismes de défense en profondeur (protections pré-authentification, cryptographie apportée, partitions FLIP/FLOP, bootloader sécurisé, etc.) du mode de démarrage DFU est décrit dans les références décrivant WooKey, nous n'y revenons donc pas dans le présent article.

4.2 Cryptographie FIDO et dérivation de clés

La cryptographie dans les tokens FIDO est (en général et pour des raisons d'optimisation) construite autour d'un secret maître K, à partir duquel sont construits les bi-clés ECDSA pour chaque service demandant un enregistrement puis une authentification.

Dans l'état de l'art des clés open source [12, 37, 54] le secret maître est en général stocké non protégé ou mal protégé [33] contre les attaques matérielles. Dans la plupart des équipements propriétaires, le secret maître est stocké dans un composant sécurisé, mais utilisable sans authentification à cause de l'absence d'authentification locale de l'utilisateur.

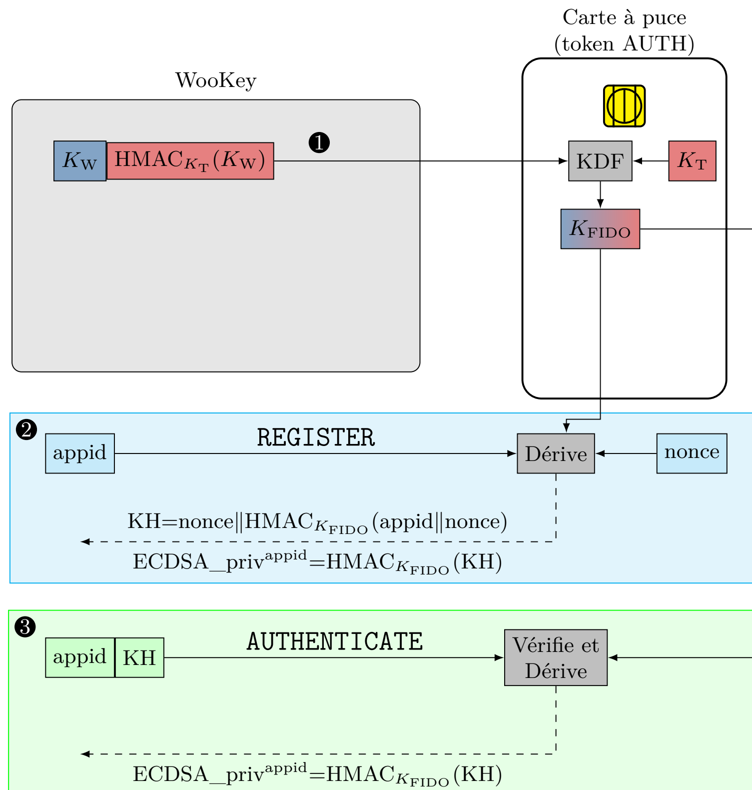

Fig. 4. Partage de secrets et dérivation

Afin de rendre plus robuste la protection de ce secret maître, nous utilisons un *partage de secrets* entre la plateforme principale WooKey et la carte à puce d'authentification externe nommée AUTH dont nous

avons étendu le code source de l'applet Javacard. Celle-ci, certifiée à un bon niveau (EAL 4+ et supérieur [56]) est changeable en cas de faiblesse découverte grâce à la portabilité Javacard [48].[4]

Il s'agit de construire la clé maître FIDO K_{FIDO}, accessible qu'après authentification de l'utilisateur et déverrouillage de la plateforme via ses PINs, à partir de secrets K_{W} et K_{T} présents indépendamment côté WooKey et côté carte à puce comme montré sur la Figure 4. Ces deux clés sont générées et provisionnées au début du cycle de vie du token FIDO lors de la mise à la clé par le SDK WooKey (permettant ainsi leur séquestre si nécessaire).

En première étape ❶, une fonction de dérivation de clé KDF [41] permet de générer K_{FIDO} et de s'assurer que cette clé ne puisse être formée sans la plateforme et la carte à puce AUTH.[5] Pour des raisons de sécurité physique, cette clé ne quitte pas la carte à puce. Afin d'éviter que la carte à puce ne soit un oracle de dérivation de clé, le HMAC de la clé de plateforme avec la clé K_{T} est vérifié.

Lorsqu'il s'agit de faire un `REGISTER` ❷ pour l'enregistrement d'un nouveau service, la plateforme envoie à la carte à puce l'`appid` (empreinte de 32 octets, haché d'une URI représentant la *Relying Party*). Celle-ci dérive alors le *Key Handle* ainsi que la clé privée ECDSA en générant un nonce aléatoire et en calculant des HMAC avec la clé K_{FIDO}. Notons que le *Key Handle* contient un HMAC pour éviter sa forge et sa malléabilité par un attaquant, le nonce permet quant à lui un non déterminisme des valeurs en fonction du service (assurant par ailleurs la possibilité d'enregistrer plusieurs comptes sur un même service).

Lorsqu'un service réalise un `AUTHENTICATE` ❸ après s'être enregistré, la plateforme envoie l'`appid` ainsi que le *Key Handle* à la carte à puce. Celle-ci commence par vérifier l'authenticité du *Key Handle* via son HMAC, et en cas de succès dérive la clé privée ECDSA *ad hoc* via un second HMAC utilisant la clé K_{FIDO}.

En conclusion, le secret K_{FIDO} n'existe que de manière éphémère et nécessite les deux clés K_{W} et K_{T} pour être créé. Pour de la défense en profondeur, nous avons aussi partagé la clé ECDSA d'attestation FIDO permettant de signer la réponse au `REGISTER` : la plateforme et la carte à puce AUTH ont chacune 16 octets sur 32 de cette clé. Un attaquant réussissant à s'emparer du token seul ou de la carte à puce seule ne

4. Nous avons d'ailleurs validé le fonctionnement du token FIDO avec plusieurs Javacards différentes de divers constructeurs en plus de la carte à puce NXP J2D081 d'origine du projet WooKey.

5. Carte à puce avec rôle d'authentification utilisateur pour WooKey.

pourra alors pas reconstituer ces éléments, et sera donc incapable de subroger à l'identité de l'utilisateur légitime. Les *PetPIN* et *UserPIN* seront les dernières barrières de sécurité dans le cas extrême où l'attaquant s'emparerait du WooKey **et** de la carte à puce.

4.3 Nouvelles tâches dans WooKey

L'implémentation du token FIDO nous oblige à revoir la répartition en tâches du mode nominal de WooKey. En effet, par manque de place en flash sur la plateforme, nous devons remplacer les tâches existantes dédiées au rôle de clé USB chiffrante par de nouvelles amenant le rôle de token FIDO/U2F (induisant *de facto* le fait qu'un utilisateur doive choisir lors de la programmation de sa plateforme le rôle qu'elle aura : clé USB chiffrante ou token FIDO). L'objectif de la présente section est de décrire les nouvelles tâches dédiées à FIDO.

Afin de limiter les risques liés aux attaques logicielles notamment depuis le PC hôte (par exemple *fuzzing* USB, avec des outils comme [61] mais également sur les couches supérieures), les services en charge des divers composants d'un token FIDO (USB, CTAP, APDU ou CBOR, cryptographie) doivent être cloisonnés afin d'assurer la confidentialité des assets FIDO (i.e. la clé K_{W} et la discussion avec la carte à puce). La logique étant qu'en cas de RCE (Remote Code Execution) dans une tâche exposée comme l'USB par exemple, l'attaquant ne puisse idéalement ni exfiltrer les secrets sensibles qui permettraient de cloner le token FIDO, ni compromettre de manière persistante la plateforme attaquée (par exemple en écrasant le firmware en flash).

Coté plateforme, comme le montre la Figure 5, notre implémentation de token FIDO est décomposée en cinq tâches ordonnancées par EwoK et qui s'exécutent en parallèle en mode nominal :

- La tâche *USB* ① se charge de l'implémentation de la pile USB jusqu'à la couche CTAPHID. Le *parsing* des commandes CTAP, respectant le standard USBHID 1.1 [2], y est effectué avant de transmettre les requêtes encapsulées via IPC à la tâche *Parser*.
- La tâche *Parser* ② est en charge d'analyser les requêtes au format APDU [4].[6] Les parseurs, et en particulier le parseur CBOR qui manipule du JSON, sont un maillon fragile souvent sources de CVE [15]. Ceux-ci sont donc, dans U2F2, cloisonnés dans une tâche sans I/O. Le *parsing* des requêtes amène à la demande d'un

6. Le *parsing* au format CBOR, lié au support de FIDO 2, sera intégré ultérieurement.

traitement FIDO. Ce traitement est effectué par la tâche *FIDO* sur demande de la tâche *Parser*.

— La tâche *FIDO* ③ est en charge des traitements cryptographiques en interactions avec la carte à puce décrits en section 4.2. Elle est également en charge, lors de requêtes FIDO, d'assurer la bonne information de l'utilisateur (protection anti-confusion) à l'aide des tâches *Storage* et *GUI*. Le mécanisme anti-confusion est décrit en section 4.4.

— La tâche *Storage* ④ est en charge de la gestion du stockage du token. Celui-ci est conservé chiffré (avec accélération matérielle `CRYP`) et intègre avec une protection anti-rejeu décrits en section 4.5. Le token se charge de stocker, pour chaque service, diverses références permettant d'assurer l'anti-confusion.

— Enfin, la tâche *GUI* ⑤ est en charge des interactions utilisateurs via l'écran et le touchscreen intégrés. C'est notamment l'interface principale gérant le *user presence* pour consentement de l'utilisateur, ainsi que le `WINK` qui est un mode particulier spécifié par FIDO pour requérir l'attention de l'utilisateur (par exemple via clignotement, etc.).

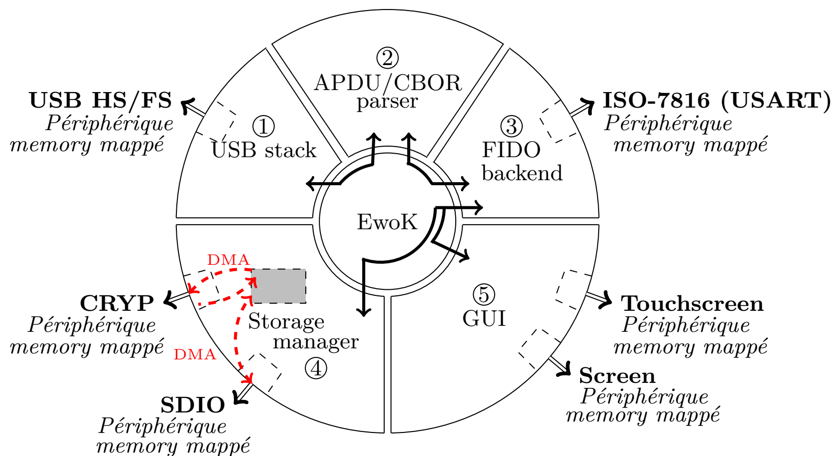

Fig. 5. Architecture en tâches du token FIDO U2F2

4.4 Système anti-confusion et « désenregistrement »

Dans le standard FIDO, le token n'a pas le nom du service pour lequel il doit construire un *Key Handle*, seul un condensat du nom de domaine (en

général une URI de la *Relying Party*) est reçu. Néanmoins, comme la phase d'enregistrement du protocole FIDO implique une action utilisateur via le *user presence*, nous avons modifié la méthode de validation utilisateur afin d'exploiter l'écran embarqué de WooKey pour permettre l'association du *Key Handle* à une icône parmi un ensemble préétabli. La référence d'icône est conservée chiffrée intègre dans la carte SD du WooKey de telle manière qu'au moment de la récupération du *Key Handle* lors de l'authentification auprès du service, ce dernier soit capable de retrouver l'icône. Pour les services connus comme Google, Facebook ou Github, nous pouvons même déduire le nom du service au travers du condensat unique associé calculé au préalable.

Ainsi, à chaque authentification FIDO, le token affichera l'icône du service que l'utilisateur a associé au moment de la phase d'enregistrement, permettant de lier de manière forte toute requête d'enregistrement venant d'un PC au service sur lequel l'utilisateur est en train d'initier la demande de connexion, réduisant fortement toute tentative de confusion.[7] Les capacités des cartes SD sont telles que ce type de fonctionnement ne pose aucun problème de volume de stockage. Les mécanismes cryptographiques induits par la phase d'authentification sont exploités pour amener du chiffrement intègre de ces éléments sur la carte SD comme décrit en section 4.5.

L'anti-confusion est permis par la collaboration de trois tâches :

— *Storage*, en charge de la gestion du stockage chiffré intègre des métadonnées FIDO (icône, nom de service, compteur anti-rejeu du service, etc.).

— *FIDO*, en charge des évènements FIDO est donc responsable d'interroger *Storage* lorsque nécessaire.

— *GUI*, en charge de fournir un retour utilisateur des métadonnées et de l'action en cours (*user presence*, etc.) via l'écran et le touchscreen.

Enfin, il est possible pour l'utilisateur de supprimer un service manuellement de la liste via l'écran tactile pour éviter toute authentification malencontreuse non voulue sur un service dont il ne veut plus l'accès. Cela permet de s'assurer qu'aucun attaquant ne pourra exploiter un service qui a mal supprimé un compte côté *Relying Party*, ou un utilisateur qui a oublié d'effectuer cette suppression sur un service qu'il n'utilise plus et qui se fait voler son token.

Afin de faciliter la gestion des services (à ajouter comme services utilisables, ou à supprimer) nous avons aussi développé en Python une

7. L'attaque n'est pas entièrement supprimée, mais fortement réduite.

application sur PC dédiée donnant un accès supplémentaire (en plus de l'écran tactile du token) à la carte SD : un aperçu de l'interface est présenté sur la Figure 6. Ainsi, il suffit à l'utilisateur d'insérer sa carte à puce AUTH et sa carte SD dans des lecteurs dédiés reliés au PC, et cette dernière est déchiffrée pour en éditer le contenu. Notons que cette application nécessite l'utilisation des clés de plateforme WooKey séquestrées sur le PC : il faut donc que ces opérations se fassent un un PC de confiance *offline* (par exemple le PC de *provisionning* et signature de firmware ainsi que mise à la clé du token FIDO).

4.5 Gestion du stockage dans U2F2

Le stockage des données dans U2F2 est effectué sur la carte microSD de la plateforme WooKey. Les données sont chiffrées et intègres, avec une clé dédiée pour chacun des usages. Le chiffrement utilise de l'AES-CBC-ESSIV hérité de la plateforme WooKey et adapté au cas d'usage du token FIDO. L'intégrité utilise un HMAC global et des HMAC locaux par slot (arbre de Merkle à deux étages) décrits ci-après. La clé de chiffrement et la clé d'intégrité sont dérivées d'une même clé maître déverrouillée au démarrage de la plateforme, stockée et envoyée par la carte à puce AUTH après l'authentification de l'utilisateur via ses deux PINs.

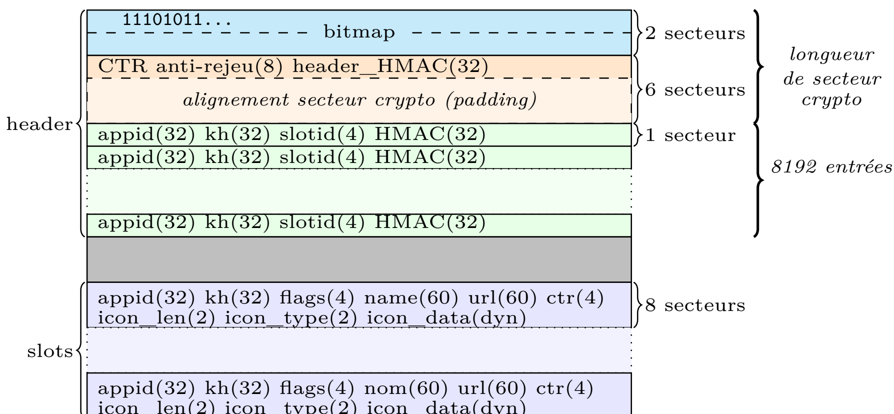

Fig. 7. Organisation du stockage externe dans U2F2

Les données sont ordonnées selon une structure voulue simple, sans notion de système de fichiers. Comme le décrit la Figure 7, le stockage est composé de deux éléments principaux. Tout d'abord, un en-tête, en charge

```
U2F2 FIDO configurator

Slot 0000, Slotid='@0x00002008', Appid='cdcdcdcdcdcdcdcdcdcdcdcdcdcdcdcdcdcdcdcdcdcdcdcdcdcdcdcdcdcdcdcd', name='Mon service custom'
Slot 0001, Slotid='@0x00002010', Appid='53a15ba42a7c0325b8dbee289634a48f58aea3246645d5ff418f9bb8819885a9', name='Keeper'
Slot 0002, Slotid='@0x00002018', Appid='d65f005ef4dea9320c9973053c95ff6020115d5fec1b7fee41a578e18df9ca8c', name='Keeper'
Slot 0003, Slotid='@0x00002020', Appid='302fd5b4492a07b9febb30e73269eca501205ccfe0c20bf7b472fa2d31e21e63', name='Bitfinex'
Slot 0004, Slotid='@0x00002028', Appid='68201915d74cb42af5b3cc5c95b9553e3e3a83b4d2a93b45fbadaa8469ff8e6e', name='Dashlane'
Slot 0005, Slotid='@0x00002030', Appid='1b3c16dd2f7c46e2b4c289dc16746bcc60dfcf0fb818e13215526e1408e7f468', name='u2f.bin.coffee'
Slot 0006, Slotid='@0x00002038', Appid='fabeece3982fad9ddcc98f91bd2e75afc7d1f4ca544929b2d0d04212dffa30fa', name='Tutanota'
Slot 0007, Slotid='@0x00002040', Appid='70617dfed065863af47c15556c91798880828cc407fdf70ae85011569465a075', name='GitHub'
Slot 0008, Slotid='@0x00002048', Appid='968978a29953de52d3ef0f0c71b7b7b6b1af9f08e257896a8d8126918530293b', name='Amazon Web Services'
Slot 0009, Slotid='@0x00002050', Appid='6966abe3674ea2f53079eb710197848c9be6f363992fd029e9898447cb9f0084', name='FastMail'
Slot 0010, Slotid='@0x00002058', Appid='12743b921297b77f1135e41fdedd4a846afe82e1f36932a9912f3b0d8dfb7d0e', name='Bitbucket'
Slot 0011, Slotid='@0x00002060', Appid='a34d309ffa28c12414b8ba6c07ee1efae1a85e8a04614859a67c0493b6956190', name='Bitwarden'
Slot 0012, Slotid='@0x00002068', Appid='f3e2042f94607da0a9c1f3b95e0d2f2bb2e069c5bb4fa764affa647d847b7ed6', name='Duo'
Slot 0013, Slotid='@0x00002070', Appid='2ac6ad09a6d0772c44da73a6072f9d240fc6854a70d79c1024ff7c7559593292', name='Stripe'
Slot 0014, Slotid='@0x00002078', Appid='a54672b222c4cf95e151ed8d4d3c767a6cc349435943794e884f3d023a8229fd', name='Google'
Slot 0015, Slotid='@0x00002080', Appid='a4e22dcafea7e90e128950113989fc45978dc9fb87767560516c1c69dfdfd196', name='Gandi'
Slot 0016, Slotid='@0x00002088', Appid='c50f8a7b708e92f82e7a50e2bdc55d8fd91a22fe6b29c0cdf7805530842af581', name='Dropbox'
Slot 0017, Slotid='@0x00002090', Appid='08b2a3d41939aa31668493cb36cdcc4f16c4d9b4c8238b73c2f672c033007197', name='Slush Pool'
Slot 0018, Slotid='@0x00002098', Appid='38804f2eff74f228b74151c201aa82e7e8eefcacfecf23fa146b13a37666314f', name='Slush Pool'
Slot 0019, Slotid='@0x000020a0', Appid='e7be96a51bd0192a72840d2e5909f72ba82a2fe93faa624f03396b30e494c804', name='GitLab'
Slot 0020, Slotid='@0x000020a8', Appid='9d61442f5ce133bd46544fc42f0a6d54c0deb88840cac2b6aefa6514f89349e9', name='Fedora'

Edit Slot 14

appid:  :f95e151ed8d4d3c767a6cc349435943794e884f3d023a8229fd
kh:     )000000000000000000000000000000000000000000000000000
name:   Google
url:    google.com
ctr:    0
flags:  0
icon

Apply modifications   Delete slot   Cancel
```

Fig. 6. GUI sur PC de gestion des services FIDO/U2F (sur la carte SD)

de fournir les informations d'intégrité et les références d'adresse de stockage pour chaque compte (service et utilisateur associé), identifié par le couple $\{appid, \texttt{SHA-256}(KeyHandle)\}$. Garder ce couple est nécessaire pour gérer le fait que pour un même service, plusieurs comptes utilisateurs peuvent être associés au même token FIDO. Seuls les *Key Handle* différencient ces situations, l'appid ne suffisant pas. Le fait de garder un condensat du *Key Handle* s'explique par le besoin de l'anonymiser sur la carte SD : si le chiffrement de celle-ci est cassé, il ne doit pas être possible d'avoir un avantage pour monter une attaque. Ensuite se trouve une liste de comptes, stockés dans des unités de stockage appelées *slots* contenant des métadonnées utiles.

Pour les besoins du PoC que nous présentons, nous nous sommes limités à 8192 slots au maximum, soit au plus 8192 couples service et compte en simultané sur la carte SD. Ce maximum peut aisément être étendu en modifiant à la marge le PoC, et les capacités des cartes SD modernes ne limitent en rien cela. Ce nombre total de comptes simultanés a cependant un impact sur les performances d'accès à la carte SD chiffrée et intègre comme discuté en section 4.6. Nous pensons néanmoins que pour des usages « classiques », et en tenant compte des « désenregistrements » dans la vie du token (donc libération des slots), ce chiffre semble raisonnable pour une utilisation nominale.

L'en-tête se décompose comme suit :

— Une bitmap listant l'ensemble des 8192 slots avec chaque bit précisant si le slot est actif (ayant déjà été enregistré) ou non.

— Un compteur anti-rejeu sur 8 octets permettant d'assurer une cohérence avec une valeur stockée dans la carte à puce AUTH. Cette valeur permet de réaliser un anti-rejeu temporel car elle est incrémentée à chaque déverrouillage complet réussi de la plateforme. En cas de détection de rejeu (juste après le déverrouillage), nous faisons le choix de prévenir l'utilisateur plutôt que de procéder à des actions plus destructives. En effet, ce rejeu peut être légitime en cas de séquestre d'une sauvegarde de la carte SD et restauration. Dans le cas où l'utilisateur acquitte ce message, le compteur est resynchronisé, sinon la plateforme redémarre.

— Un HMAC, sur l'ensemble des éléments critiques de l'en-tête (la bitmap et l'ensemble des entrées de slots actifs).

Pour des raisons de performances, seules les données utiles de l'en-tête sont vérifiées. Cela n'induit pas de problème de sécurité vu que les autres données ne sont pas lues ni interprétées. La table des références de slots actifs est écrite dans les premiers secteurs de la carte SD. Chaque

entrée de la table référence le secteur (au sens secteur SD, soit 512 octets) hébergeant le slot, ainsi que son motif d'intégrité HMAC. Les slots sont écrits en séquence, avec une taille fixe de 4096 octets chacun. Les deux niveaux de HMAC dans le header et par slot permettent une gestion d'intégrité globale flexible, à l'image des arbre de Merkle (c'est en fait un arbre simple à deux niveaux).

Chaque slot possède :

— Le condensat appid envoyé par le Browser.
— Le condensat du *Key Handle* (gestion de plusieurs comptes sur un même service).
— L'URL du service choisie par l'utilisateur.
— Un nom représentatif configurable.
— Le compteur d'anti-rejeu d'authentification FIDO (`CTR`) spécifique à ce compte, dont l'intérêt est discuté en section 4.6.
— Une icône ou une couleur, facilement reconnaissables.
— Des *flags* liés à la configuration et à l'état du service (sensibilité du service, verrouillage, *user presence* forcé, etc.). Ils sont prévus pour une extension d'usages futurs en fonction des contextes d'utilisation du token. On peut par exemple imaginer des verrouillages temporaires de services (à ne pas confondre avec un « désenregistrement ») mis en suspens pour être réactivés plus tard, etc.

4.6 Évaluation de U2F2

La séparation cryptographique permettant de générer et d'utiliser K_{FIDO} décrite en 4.2, tout comme l'usage d'un média de stockage, ont nécessairement un impact en termes de performances, de réactivité, et éventuellement de conformité du fait de la divergence aux implémentations canoniques de tokens FIDO « classiques ». Les latences du protocole de discussion avec la carte à puce ainsi qu'avec la carte SD s'ajoutent aux opérations FIDO brutes. L'objectif de la présente section est d'analyser l'impact de notre architecture à la fois sur la conformité et sur les performances.

Conformité au standard FIDO/U2F : Nous n'avons malheureusement pas eu accès aux outils de conformité fournis par l'alliance FIDO [22], car ils nécessitent une procédure d'enregistrement requérant un `VendorID` USB. Nous avons cependant à la fois utilisé les outils implémentés par l'équipe SoloKey [55] et nos propres outils, permettant de valider la conformité, l'endurance et les performances de notre PoC. Le token U2F2 valide

l'ensemble des tests correspondant à une compatibilité FIDO 1.2. Ces tests sont classés en deux familles :

— La conformité du transport qui vérifie la pile CTAPHID et les commandes de niveau CTAP.
— La conformité de la pile U2F qui vérifie la bonne prise en compte des requêtes FIDO, incluant des séquences d'enregistrement et d'authentification auprès du token.

Au delà de la conformité, nous avons aussi effectué des tests d'endurance du token en développant des scripts permettant d'effectuer des milliers de cycles de `REGISTER` et `AUTHENTICATE` avec des appid divers (en émulant le *user presence* côté token pour automatiser cela). Cela nous a permis de nous assurer de la robustesse de la partie cryptographique, et de son interopérabilité avec les piles U2F existantes.

Performances du token : Plusieurs opérations sont effectuées au démarrage de la plateforme et ont donc un « coût fixe ». Notamment, la dérivation de la clé K_{FIDO} (étape ❶ de la section 4.2) et la récupération de la demi-clé privée d'attestation sont faites juste après la saisie des PINs, ainsi que la récupération du compteur anti-rejeu stocké dans la carte à puce. Nous avons optimisé l'intégration de ces échanges avec la Javacard au démarrage de WooKey, ajoutant ainsi quelques centaines de millisecondes au cycle de déverrouillage du *device*, ce qui est quasiment imperceptible du point de vue de l'utilisateur (ce cycle prenait déjà quelques secondes qui plus est « masquées » par les interactions de saisies des PINs). Le seul élément pouvant ajouter une durée significative est la vérification d'intégrité et de l'anti-rejeu de la carte SD qui demande un nombre de cycles CPU proportionnel au nombre de comptes FIDO (slots) actifs : ces éléments sont discutés plus en détail ci-après.

Une fois la plateforme démarrée, chaque opération de `REGISTER` et `AUTHENTICATE` nécessitent chacun un échange avec la carte à puce pour récupérer un *Key Handle* et une clé privée (étape ❷ de la section 4.2) ou juste une clé privée (étape ❸). La plateforme effectue ensuite une signature ECDSA pour envoyer sa réponse au browser. Ces opérations coûtent à peu près *une seconde*[8] supérieur aux tokens existants qui les font en quelques dizaines à centaines de millisecondes. Cela s'explique par notre architecture de dérivation de clés dans le composant sécurisé de la carte à puce ainsi que les contre-mesures mises sur l'implémentation ECDSA de la plateforme pour plus de sécurité. Notons qu'à l'usage, et du

8. Environ 500 ms pour l'échange avec la Javacard, et environ 400 ms pour la signature ECDSA et le formatage de la réponse FIDO sur la plateforme.

fait du *user presence*, cette attente est très raisonnable et ne nuit pas à l'expérience utilisateur.

En plus des opérations cryptographiques directement liées à FIDO, l'accès à la carte SD peut rajouter un peu de temps supplémentaire (du fait du chiffrement intègre des slots). Grâce à l'utilisation d'un arbre de Merkle, les accès coûteux sont la vérification d'intégrité du header de la carte SD (au démarrage seulement, donc), ainsi que les écritures pour lesquelles il faut modifier le HMAC du slot modifié ainsi que le HMAC du header. Lors des opérations de `REGISTER` et `AUTHENTICATE` sur des comptes existants, il faut un accès en lecture au compte associé (peu coûteux donc), mais il faut aussi un accès en écriture pour gérer le compteur anti-rejeu FIDO associé au compte et l'incrémenter à chaque fois. Lors de la gestion d'un compte (administration de ses métadonnées ou suppression), l'opération est coûteuse mais ces opération arrivent rarement.

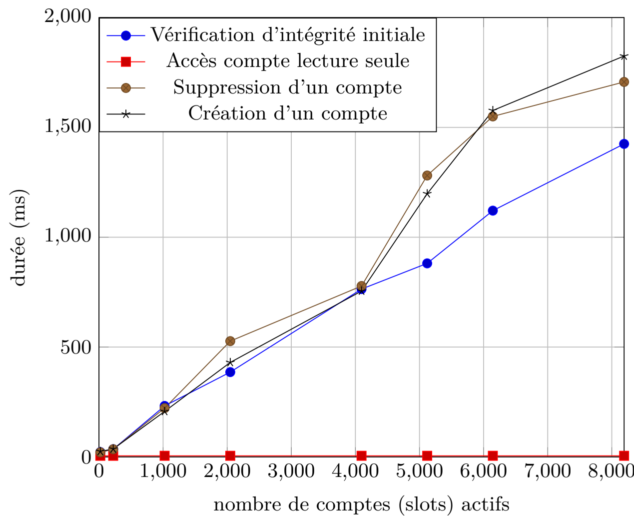

Fig. 8. Performances des opérations sur la carte SD en fonction du nombre de services actifs

Le temps des opérations « coûteuses » (nécessitant un recalcul du motif d'intégrité du header) est proportionnel au nombre de slots et est représenté sur la Figure 8. En dessous de 2000 services enregistrés le temps d'accès de 500 millisecondes reste très raisonnable, mais une latence de plus d'une seconde se fait ressentir au delà de 5000 slots. L'opération de

lecture seule sans modification d'intégrité a de manière logique une latence constante et faible de 5 millisecondes.

Au final, et de manière pragmatique, ces latences restent imperceptibles à l'usage pour un nombre de comptes actifs en dessous du millier, et commencent à avoir un impact après. Nous soulignons néanmoins le fait que pour une utilisation standard, un millier de comptes simultanés reste une limite raisonnable.

L'optimisation de ces temps d'accès (notamment en utilisant de l'accélération matérielle pour le HMAC, pour l'instant non utilisée, ou le passage à de l'AES-GCM accéléré) fait partie des améliorations. Le calcul en logiciel du HMAC prend en effet au moins 50% du temps de vérification et modification des slots. Notons enfin que l'impact non négligeable sur la latence des `REGISTER` et `AUTHENTICATE` avec un nombre de comptes élevé est entre autres lié à l'utilisation d'un compteur dédié à chaque compte/service, choix que nous avons fait pour des raisons de sécurité. L'utilisation d'un compteur unique global (par exemple stocké en flash interne de la plateforme ou dans la carte à puce) enlèverait toute contrainte de mise à jour des motifs d'intégrité lors de `REGISTER` et `AUTHENTICATE`, au détriment de la sécurité comme discuté ci-après.

Compteur anti-rejeu par service : La plupart des tokens FIDO qui existent possèdent un compteur anti-rejeu global à tous les services, afin d'éviter un stockage de données associées à chaque compte (cela simplifie grandement la gestion du stockage sur le token). La logique implémentée côté service est alors « laxiste » car elle ne vérifie que si ce compteur est strictement supérieur au compteur précédemment utilisé (stocké côté service) : il n'est pas possible de savoir si des demandes d'`AUTHENTICATE` ont été effectuées sur un *Key Handle* donné via une usurpation malveillante du service (locale sur le poste de l'utilisateur ou distante). Pour des services *sensibles*, il serait plus logique d'implémenter une vérification stricte du compteur pour s'assurer que celui-ci ne peut être qu'incrémenté entre deux `AUTHENTICATE` : c'est dans ce cadre qu'un compteur local tel qu'implémenté dans U2F2 prend tout son sens.[9]

9. Cela ne concerne évidemment que les *Relying Party* dont l'implémentation FIDO est adaptable, par exemple dans des contextes d'accès à des mails professionnels sur une infrastructure maîtrisée.

5 Conclusion

Nous avons présenté dans cet article le concept de second facteur d'authentification 2FA tel que spécifié par le consortium FIDO, notamment via sa version 1.2 U2F. Si l'on s'intéresse à la sécurité du token d'authentification en lui-même, plusieurs limitations non couvertes par cette spécification ou laissées au choix de l'implémentation amènent des risques de sécurité dans des contextes sensibles. Dans des scénarios de vol, vol avec remise, piégeage et attaques matérielles (canaux auxiliaires ou en fautes) ou logicielles (exploitation de Run Time Errors), la sécurité de la majorité des tokens open source, voire même propriétaires, laisse à désirer ou est difficilement auditable.

Nous avons donc détaillé les diverses contre-mesures apportées par notre preuve de concept U2F2 implémentée sur une plateforme WooKey, et bénéficiant donc des couches de défense déjà présentes. L'intégration des spécificités de FIDO a aussi amené son lot de réflexions pour aboutir à une architecture de sécurité séparée entre la plateforme et la carte à puce externe.

Concernant les travaux en cours, nous nous concentrons sur l'intégration de modules pour le support de FIDO 2.0 et 2.1, notamment le parser CBOR et compléter la pile pour couvrir tout CTAP. L'utilisation d'autres canaux comme le NFC ou le Bluetooth sont aussi des pistes suivies. La capacité à enrichir l'usage FIDO *grand public* en un usage professionnel, intégrant la notion de flotte et de révocation de token tout en respectant le standard, est également en cours de développement.

Enfin, l'intégration de modules annexes comme l'OTP, le PGP ou le KeePass généralement présents sur les tokens FIDO du commerce, est aussi prévue sur le plus long terme.

Références

1. Client to Authenticator Protocol (CTAP). `https://fidoalliance.org/specs/fido-v2.0-rd-20180702/fido-client-to-authenticator-protocol-v2.0-rd-20180702.html`.
2. Device Class Definition for Human Interface Devices (HID). `https://www.usb.org/sites/default/files/documents/hid1_11.pdf`.
3. FIDO Alliance. `https://fidoalliance.org/`.
4. FIDO U2F 1.2. `https://fidoalliance.org/specs/fido-u2f-v1.2-ps-20170411/`.
5. FIDO U2F HID Protocol Specification. `https://fidoalliance.org/specs/fido-u2f-v1.2-ps-20170411/fido-u2f-hid-protocol-v1.2-ps-20170411.html`.

6. FIDO U2F Raw Message Formats. https://fidoalliance.org/specs/fido-u2f-v1.2-ps-20170411/fido-u2f-raw-message-formats-v1.2-ps-20170411.html.
7. FreeOTP. https://freeotp.github.io/.
8. Google Authenticator. https://www.google-authenticator.com/.
9. Hashcat advanced password recovery. https://hashcat.net/hashcat/.
10. Kaspersky password checker. https://password.kaspersky.com/fr/.
11. La clé YubiKey. https://www.yubico.com/la-cle-yubikey/?lang=fr.
12. Titan Security Key. https://cloud.google.com/titan-security-key?hl=fr.
13. TockOS. https://www.tockos.org/.
14. Tokens matériels RSA SecurID. https://www.rsa.com/fr-fr/products/rsa-securid-suite/rsa-securid-access/securid-hardware-tokens/rsa-securid-hardware-tokens.
15. CVE-2019-9403 : Out-of-Bound due to imporper casting in cn-cbor. https://www.cvedetails.com/cve/CVE-2019-9403/, 2019.
16. CVE-2020-10663 : Unsafe Object Creation Vulnerability in JSON. https://nvd.nist.gov/vuln/detail/CVE-2020-10663, 2020.
17. Inter-CESTI : Methodological and Technical Feedbacks on Hardware Devices Evaluations. https://www.sstic.org/2020/presentation/inter-cesti_methodological_and_technical_feedbacks_on_hardware_devices_evaluations/, 2020.
18. Ledger Nano X. https://shop.ledger.com/products/ledger-nano-x, 2020.
19. OnlyKey Fall 2020 Update. https://crp.to/2020/10/01/onlykey-fall-2020-update/, 2020.
20. The New Nitrokey 3 With NFC, USB-C, Rust, Common Criteria EAL 6+. https://www.nitrokey.com/news/2021/new-nitrokey-3-nfc-usb-c-rust-common-criteria-eal-6, 2021.
21. Trussed Announcement. https://trussed.dev/blog/trussed-announcement/, 2021.
22. FIDO Alliance. Conformance Self-Validation Testing. https://fidoalliance.org/certification/functional-certification/conformance/.
23. Davit Baghdasaryan, Brad Hill, Joshua E Hill, and Douglas Biggs. Fido security reference. 2013.
24. Manuel Barbosa, Alexandra Boldyreva, Shan Chen, and Bogdan Warinschi. Provable Security Analysis of FIDO2. 2020.
25. Ryad Benadjila, Cyril Debergé, Patricia Mouy, and Philippe Thierry. From CVEs to proof : Make your USB device stack great again. In *SSTIC*, 2021.
26. Ryad Benadjila, Arnauld Michelizza, Mathieu Renard, Philippe Thierry, and Philippe Trebuchet. WooKey : designing a trusted and efficient USB device. In *Proceedings of the 35th Annual Computer Security Applications Conference*, pages 673–686, 2019.
27. Ryad Benadjila, Mathieu Renard, Philippe Trebuchet, Philippe Thierry, Arnauld Michelizza, and Jérémy Lefaure. WooKey : USB devices strike back. In *Symposium sur la sécurité des technologies de l'information et des communications*, volume 25, 2018.

28. Ryad Benadjila, Mathieu Renard, Philippe Trebuchet, Philippe Thierry, Arnauld Michelizza, and Jérémy Lefaure. Wookey : Usb devices strike back. "https://wookey-project.github.io/index.html", 2018.
29. Mickaël Bergem and Florian Maury. A first glance at the U2F protocol. *SSTIC 2016*.
30. C. Bormann and P. Hoffman. Concise Binary Object Representation (CBOR). RFC 7049 (Proposed Standard), October 2013.
31. T. Bray. The JavaScript Object Notation (JSON) Data Interchange Format. RFC 7158 (Proposed Standard), March 2014. Obsoleted by RFC 7159.
32. Elie Bursztein. The bleak picture of two-factor authentication adoption in the wild. https://elie.net/blog/security/the-bleak-picture-of-two-factor-authentication-adoption-in-the-wild/, 2018.
33. Simon Collin. *Side channel attacks against the Solo key - HMAC-SHA256 scheme*. PhD thesis, UCL - Ecole polytechnique de Louvain, 2020.
34. Cybernews. COMB : largest breach of all time leaked online with 3.2 billion records. https://cybernews.com/news/largest-compilation-of-emails-and-passwords-leaked-free/.
35. Ledger Donjon. Unfixable Key Extraction Attack on Trezor. https://donjon.ledger.com/Unfixable-Key-Extraction-Attack-on-Trezor/.
36. Haonan Feng, Hui Li, Xuesong Pan, and Ziming Zhao. A Formal Analysis of the FIDO UAF Protocol. 2021.
37. Sergei Glushchenko (gl sergei). Project : u2f-token, may 2019. https://github.com/gl-sergei/u2f-token.
38. Google. OpenSK. https://github.com/google/OpenSK, 2020.
39. Christopher Harrell. Getting a biometric security key right. https://www.yubico.com/blog/getting-a-biometric-security-key-right/, 2020.
40. Kraken. Kraken Identifies Critical Flaw in Trezor Hardware Wallets. https://blog.kraken.com/post/3662/kraken-identifies-critical-flaw-in-trezor-hardware-wallets/.
41. H. Krawczyk and P. Eronen. HMAC-based Extract-and-Expand Key Derivation Function (HKDF). RFC 5869 (Informational), May 2010.
42. Ledger. Ledger bolos. https://www.ledger.fr/2016/03/02/introducing-bolos-blockchain-open-ledger-operating-system/, 2017.
43. Amit Levy, Bradford Campbell, Branden Ghena, Daniel Giffin, Pat Pannuto, Prabal Dutta, and Philip Levis. Multiprogramming a 64 kB Computer Safely and Efficiently. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP 2017)*, October 2017.
44. Victor LOMNE and Thomas ROCHE. A Side Journey to Titan. Cryptology ePrint Archive, Report 2021/028, 2021. https://eprint.iacr.org/2021/028.
45. Matus Nemec, Marek Sys, Petr Svenda, Dusan Klinec, and Vashek Matyas. The Return of Coppersmith's Attack : Practical Factorization of Widely Used RSA Moduli. Technical Report CVE-2017-15361, oct 2017.
46. Colin O'Flynn. Min()imum failure : EMFI attacks against USB stacks. In *13th USENIX Workshop on Offensive Technologies (WOOT 19)*, Santa Clara, CA, August 2019. USENIX Association.

47. Openwall. John the Ripper password cracker. https://www.openwall.com/john/.
48. Oracle. Java card 3 platform runtime environment specification, classic edition version 3.0.5, 2015.
49. Christoforos Panos, Stefanos Malliaros, Christoforos Ntantogian, Angeliki Panou, and Christos Xenakis. A security evaluation of FIDO's UAF protocol in mobile and embedded devices. In *International Tyrrhenian Workshop on Digital Communication*, pages 127–142. Springer, 2017.
50. Gwendal Patat and Mohamed Sabt. Please Remember Me : Security Analysis of U2F - Remember Me Implementations in The Wild. *SSTIC2020*.
51. Manini Roy. Microsoft's Path to Passwordless - FIDO Authentication for Windows & Azure Active Directory, 2018.
52. John Scott-Railton and Katie Kleemola. Two-Factor Authentication Phishing from Iran, August 2015. https://citizenlab.ca/2015/08/iran_two_factor_phishing/.
53. Kelly Sheridan. Younger Generations Drive Bulk of 2FA Adoption. https://www.darkreading.com/application-security/younger-generations-drive-bulk-of-2fa-adoption/d/d-id/1336581.
54. Solokeys. Project : solokeys/solo, may 2020. https://github.com/solokeys/solo.
55. Solokey team. fido2-tests. https://github.com/solokeys/fido2-tests, 2021.
56. The Common Criteria Project. The Common Criteria Portal. https://www.commoncriteriaportal.org/.
57. Craig Timberg. German researchers discover a flaw that could let anyone listen to your cell calls. *The Washington Post*, 2014.
58. Nikolaos Tsalis and Dimitris Gritzalis. Hacking web intelligence : Open source intelligence and web reconnaissance concepts and techniques, sudhanshu chauhan, nutan kumar panda, elsevier publications, USA (2015). *Comput. Secur.*, 55 :113, 2015.
59. Vice. A Hacker Got All My Texts for $16. https://www.vice.com/en/article/y3g8wb/hacker-got-my-texts-16-dollars-sakari-netnumber.
60. Trezor Wiki. Trezor U2F. https://wiki.trezor.io/U2F.
61. Grzegorz Wypych. usb-tester. https://github.com/h0rac/usb-tester, 2020.
62. Yubico. Security advisory 2017-10-16 – Infineon weak RSA key generation. Technical Report YSA-2017-01, oct 2017.