POLITECNICO DI TORINO

DIPARTIMENTO DI AUTOMATICA E INFORMATICA

Corso di Laurea Magistrale in
Ingegneria Informatica

Tesi di Laurea Magistrale

Libreria di comunicazione tra OpenSSH ed autenticatore FIDO Bluetooth

Relatore
Prof. Enrico Magli

Candidato
Davide Sordi

Anno Accademico 2020-2021

Sommario

Negli ultimi anni i servizi online offerti dalle aziende sono cresciuti esponenzialmente, questo ha portato ogni utente a dover gestire un numero sempre più elevato di credenziali di accesso. Le password, in uso da anni stanno rivelando i propri limiti dovuti sia all'inesperienza dell'utente, che a una cattiva gestione da parte dei fornitori di servizi ad esempio: l'utilizzo di password deboli o il salvataggio delle stesse in chiaro senza meccanismi di protezione. L'obiettivo della FIDO Alliance e dello standard FIDO2 è l'eliminazione delle password e lo sfruttamento di metodi di autenticazione più sicuri e meno impegnativi per l'utente. Lo standard si basa sull'utilizzo di un dispositivo sicuro di autenticazione, sia esso uno smartphone o una chiavetta di sicurezza USB. Attraverso il meccanismo della cifratura asimmetrica, è possibile generare una coppia di chiavi: una pubblica che sarà inviata ai server presso i quali ci si vuole registrare e una privata che sarà custodita e protetta all'interno dell'autenticatore. Per mezzo della chiave privata, il dispositivo effettuerà la cifratura di un messaggio generato dal server, il quale, decifrandolo con la chiave pubblica sarà in grado di determinare l'identità dell'utente. Recentemente, la suite di comunicazione OpenSSH ha introdotto il supporto all'autenticazione mediante autenticatori FIDO fornendo di base una libreria di comunicazione per le chiavette di sicurezza USB e lasciando delle linee guida per l'implementazione delle controparti per Bluetooth Low Energy e NFC. L'obiettivo di questo lavoro è l'implementazione in linguaggio C++ della libreria di comunicazione tra la suite di OpenSSH e un autenticatore Bluetooth. In particolare, è stato utilizzato uno smartphone Android che sfrutta la tecnologia di ToothPic, una startup italiana con sede all'interno dell'I3P, l'incubatore del Politecnico di Torino che si occupa di cybersecurity. ToothPic ha prodotto un'applicazione per smartphone conforme allo standard di autenticazione FIDO2 che rende il dispositivo un token di

sicurezza in grado di autenticare l'utente in modo sicuro e senza necessità di dover scrivere o memorizzare alcuna password. L'autenticatore certificato FIDO2 è basato su una tecnologia proprietaria per la protezione delle credenziali generate basata sul rumore che ogni sensore fotografico lascia quando scatta una fotografia. Questa impronta digitale è una proprietà derivante direttamente dal wafer di silicio utilizzato per produrre il sensore e risulta quindi non clonabile e non modificabile e soprattutto diversa per ogni fotocamera. L'autenticatore sfrutta questa proprietà per offuscare la chiave privata che viene generata in fase di registrazione e successivamente de offuscarla in fase di firma del messaggio. Il protocollo di comunicazione tra l'autenticatore e il client OpenSSH è definito nello standard denominato CTAP2 (Client To Authenticator Protocol) in particolare, sulla versione 2.0 si basa l'autenticatore utilizzato.

Ringraziamenti

Prima di procedere con la trattazione, vorrei ringraziare il mio relatore, il Prof Magli Enrico per l'opportunità di tesi e l'assistenza durante la redazione.

Vorrei, in aggiunta, ringraziare Palazzi Cristiano e Coluccia Giulio di ToothPic per il supporto fornitomi durante il periodo di tirocinio e la successiva stesura della tesi.

Inoltre, uno speciale ringraziamento va alla mia famiglia, mia mamma, mia sorella e i miei nonni che mi hanno supportato lungo tutto il percorso e che hanno sempre creduto in me.

Infine, un grande grazie a Greta e alla sua famiglia.

Indice

Elenco delle figure 7

1 Introduzione 8

2 Background 11
2.1 Fido Alliance e FIDO2 11
2.1.1 WebAuthn 12
2.1.2 CTAP2 12
2.1.3 I problemi delle password e la soluzione passwordless 12
2.1.4 Autenticatori, hardware e software 14
2.2 Crittografia a chiave pubblica 15
2.2.1 Registrazione 15
2.2.2 Firma 16
2.3 ToothPic 16
2.3.1 La Tecnologia ToothPic 17
2.3.2 L'autenticatore e la certificazione 18
2.4 OpenSSH 19
2.4.1 SSH-KEYGEN 20
2.4.2 SSH-AGENT 20
2.4.3 SSH-ADD 21
2.5 Bluetooth Low Energy (BLE) 21
2.5.1 Ruoli: Client e Server 22
2.5.2 UUIDs 22
2.5.3 GATT, Servizi e Caratteristiche 22

3 Progettazione della libreria BLESSH 25
3.1 Introduzione e prerequisiti 25
3.1.1 I file PROTOCOL.u2f e sk-api.h 26

3.2 La scelta delle librerie 27
3.2.1 Libreria BLE 27
3.2.2 Libreria CBOR 28
3.3 BLESSH e progetto per testing 28

4 Sviluppo della libreria BLESSH 29
4.1 Strutture dati e prototipi da sk-api.h 29
4.1.1 Struttura sk_enroll_response 30
4.1.2 Struttura sk_sign_response 31
4.1.3 Struttura sk_resident_key 32
4.1.4 Struttura sk_option 32
4.1.5 Prototipo sk_api_version 33
4.1.6 Prototipo sk_enroll 33
4.1.7 Prototipo sk_sign 34
4.1.8 Prototipo sk_load_resident_keys 35
4.2 Implementazione dei metodi della libreria 36
4.2.1 Funzione sk_api_version 36
4.2.2 Funzione sk_enroll 37
4.2.3 Funzione sk_sign 42
4.2.4 Funzione sk_load_resident_keys 44
4.3 Testing della libreria 47
4.3.1 Comandi OpenSSH 47

5 Conclusioni 51
5.1 Limiti e futuri sviluppi 51

Bibliografia 55

Elenco delle figure

2.1 Schema semplice WebAuthn + CTAP. 13
2.2 La tecnologia ToothPic. 18
2.3 Logo autenticatore certificato dalla FIDO Alliance. 19
3.1 Schema di comunicazione OpenSSH - BLESSH - FIDO. . . 26
4.1 Composizione della struttra dati Authenticator Data . . . 31
4.2 Operazione di firma da parte dell'autenticatore 32
4.3 Diagramma del flusso di esecuzione della funzione di enroll. 37
4.4 Diagramma del flusso di esecuzione della funzione di sign. . 43
4.5 Diagramma del flusso di esecuzione della funzione di load. 44

Capitolo 1

Introduzione

Le password sono un metodo di autenticazione che tutti conoscono, sono in uso da decine di anni e tutti noi ne abbiamo sicuramente più di una. Tuttavia, negli ultimi anni ci si sta rendendo conto che l'utilizzo di questo metodo inizia a essere non più sufficiente a garantire un elevato standard di sicurezza. Si basti immaginare lo sforzo necessario per il mantenimento (creazione di password sicure e diverse, salvataggio sicuro, reset e cambio) dell'elevato numero di credenziali che ognuno di noi oggi possiede per accedere ai vari servizi su internet, oppure, alla facilità con cui queste possono essere sottratte mediante attacchi di phishing o ancora data breach di servizi non sicuri in cui vengono salvate in chiaro.

Ecco che quindi entra in gioco la Fido Alliance con FIDO2 che mira a diffondere uno standard aperto di autenticazione senza password basato su un dispositivo autenticatore, nel nostro caso uno smartphone. ToothPic, l'azienda presso la quale ho sviluppato il mio lavoro di tesi, ha prodotto un'applicazione per smartphone conforme allo standard di autenticazione FIDO2 che rende il dispositivo un token di sicurezza in grado di autenticare l'utente in modo sicuro e senza necessità di dover scrivere o memorizzare alcuna password.

OpenSSH, invece, è una suite di programmi open source che permette la comunicazione tra computer attraverso una sessione sicura mediante metodi tra i quali password e autenticazione a chiave pubblica. Dalla versione 8.2 di OpenSSH è supportata l'autenticazione passwordless tramite autenticatore FIDO come, ad esempio, una chiave di sicurezza USB.

L'obiettivo di questo lavoro è l'implementazione di una libreria che permetta la comunicazione tra OpenSSH e l'autenticatore sviluppato da

ToothPic conforme allo standartd FIDO tramite Bluetooth Low Energy. OpenSSH, nella sua versione base, fornisce l'implementazione della libreria equivalente per la comunicazione attraverso USB, mentre le versioni BLE e NFC (Near Field Communication) non sono ancora state implementate.

Capitolo 2

Background

In questo capitolo verranno fornite delle nozioni base necessarie a elencare e comprendere il funzionamento dei vari software e le tecnologie utilizzate.

2.1 Fido Alliance e FIDO2

La FIDO Alliance è un'associazione formata da varie aziende del settore tecnologico quali Amazon, Google, Intel e molte altre, che ha come obiettivo lo sviluppo e la promozione di uno standard aperto di autenticazione senza l'utilizzo delle password mirando quindi a evitare tutti i problemi che derivano dal loro utilizzo.

La FIDO Alliance, inoltre, si occupa di certificare i dispositivi di autenticazione mediante una suite di test standard volti a garantire la conformità dell'autenticatore con lo standard FIDO.

FIDO2 [6] (Fast Identity Online) è l'ultimo standard prodotto per garantire un'autenticazione sicura nel World Wide Web, ed è composto da due specifiche principali: WebAuthn e CTAP2[1].

[1] 2.1

2.1.1 WebAuthn

Web Authentication (WebAuthn) è una delle componenti principali del set di specifiche FIDO2, è una web-based API (Application Programming Interface) che consente ai siti web e alle applicazioni di implementare il meccanismo di login basato su FIDO2 mediante l'utilizzo di un autenticatore conforme allo standard.

WebAuthn è stato standardizzato dal W3C a marzo 2019, diventando così di fatto uno standard riconosciuto a livello globale per l'autenticazione sicura nel World Wide Web. W3C è infatti la sigla del World Wide Web Consortium, un'organizzazione che si occupa di standardizzare le tecnologie al fine di "portare il World Wide Web al massimo del suo potenziale". I più importanti browser quali Google Chrome, Microsoft Edge e Mozilla supportano le webauthn API.

2.1.2 CTAP2

Client To Authenticator Protocol (CTAP2) è l'altro tassello fondamentale dello standard FIDO2, è la tecnologia complementare al WebAuthn e si occupa di definire un protocollo di comunicazione tra un autenticatore esterno, sia esso USB, BLE o NFC, e i browser che supportano WebAuthn. Permette inoltre la comunicazione tra l'autenticatore e un'applicazione come ad esempio OpenSSH. Il CTAP2 è uno standard in evoluzione, faremo riferimento principalmente alla versione 2.0 (ultima versione "proposed standard" 30/01/2019) in quanto l'autenticatore ToothPic implementa in toto questa versione, ma per una particolare funzionalità richiesta da OpenSSH faremo riferimento alla versione 2.1, la quale tuttavia, è ancora in fase di draft. Un maggiore approfondimento al protocollo avverrà in seguito.

2.1.3 I problemi delle password e la soluzione passwordless

Come anticipato, uno degli obiettivi della FIDO Alliance è la promozione di uno standard di autenticazione che non preveda l'uso delle password. Queste sono il più semplice meccanismo di autenticazione in quanto a semplicità d'uso, tutti ne hanno almeno una. Tuttavia, le password hanno parecchi lati negativi, e risultano prede di diversi tipi di attacchi sia per

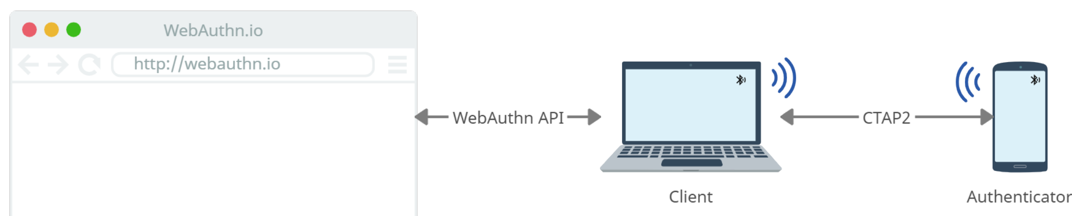

Figura 2.1. Schema semplice WebAuthn + CTAP.

colpa degli utenti che per colpa dei gestori dei servizi. Vediamo alcuni dei principali rischi:

- Password troppo semplici, secondo NordPass [20] nel 2020, nei primi 4 posti della speciale classifica delle peggiori password troviamo ad esempio“123456”, “123456789” e“password”.
- Insicurezza nella protezione delle password come, ad esempio, la loro scrittura su carta o salvataggio su documenti di testo non protetti.
- Riuso delle password per accedere a servizi diversi, mettendo così a rischio i dati personali di molteplici servizi. Secondo uno studio del Ponemon Institute [23] sia gli utenti che il personale intervistato hanno riutilizzato le proprie password in circa dieci account mediamente.
- Condivisione di username e password sia di account personali che aziendali con altre persone. Un report del Ponemon [22] Institute del 2020 ha rivelato che il 49% dei tecnici intervistati e il 51% degli utenti hanno condiviso le proprie password con i propri colleghi.
- Attacchi di phishing, questo tipo di attacco mira a rubare le credenziali tramite link di login ingannevoli inviati solitamente via mail agli utenti, da un report di Verizon [21] del 2017 si stima che circa il 30% delle mail ingannevoli vengono aperte dagli utenti e che oltre il 7% contiene file maligni che portano all’installazione di malware oltre che al furto di credenziali.
- Costi del supporto tecnico in casi di password perse. In caso di password dimenticata o persa si finisce spesso con il dover contattare il

supporto tecnico. Secondo un report di McKinsey [18] il reset delle password occupa fino al 6% delle attività degli help desks con un costo per le grandi aziende stimato tra i 5 e i 20 milioni di dollari all'anno. Nel mese di luglio del 2017 Microsoft [19] ha dovuto effettuare il reset di 686 mila password dimenticate, risultando in una spesa complessiva di oltre 12 milioni di dollari.

- Salvataggio delle password all'interno di database senza un meccanismo di hashing o peggio ancora in chiaro. Secondo la "OWASP (Open Web Application Security Project) top ten" [10] il secondo rischio per le web application è: Broken Authentication" ovvero un malfunzionamento dovuto a una mal progettazione o vulnerabilità alle funzioni che si occupano di gestire le sessioni e l'autenticazione degli utenti.

L'obiettivo quindi di FIDO2 è eliminare le password e i relativi rischi ad esse associati, sollevando la responsabilità degli utenti in quanto non dovranno più preoccuparsi di creare e mantenere password con elevati standard di sicurezza. L'utente non dovrà fare altro che scegliere uno username o una mail e poi confermare la registrazione o il login attraverso il proprio autenticatore, sia esso uno smartphone o una chiave di sicurezza usb.

2.1.4 Autenticatori, hardware e software

Come possiamo vedere dall'elenco degli autenticatori conformi allo standard FIDO2 [5], è possibile suddividerli in due principali categorie, hardware e software. Tra i dispositivi fisici i più diffusi sono quelli USB, hanno le sembianze di una tipica chiavetta USB ed hanno al loro interno dei chip di sicurezza capaci di generare, utilizzare (come vedremo in seguito) e solitamente salvare al loro interno le coppie di chiavi necessarie per l'autenticazione. Tra i più conosciuti possiamo citare le YubiKey di Yubico o le Titan di Google. Oltre a questi dispositivi di sicurezza esistono anche gli autenticatori software, solitamente delle applicazioni per smartphone, o dei programmi per computer. La sicurezza del loro funzionamento si basa sull'utilizzo del TPM (Trusted Platform Module) su pc e del TEE (Trusted Execution Environment) su smartphone, dei chip deputati alla gestione ed elaborazione di chiavi ed esecuzione degli algoritmi di cifratura. In questo caso il più famoso è Windows Hello di Microsoft. ToothPic,

ha sviluppato un autenticatore software disponibile sia per Android che iOS che sfrutta una tecnologia proprietaria per la protezione delle chiavi, ma prima di parlarne più approfonditamente occorre capire la distinzione tra chiave pubblica e privata.

2.2 Crittografia a chiave pubblica

La crittografia a chiave pubblica detta anche crittografia asimmetrica è un meccanismo di sicurezza basato su una coppia di chiavi crittografiche: la chiave privata e la chiave pubblica. Come si evince dal nome, la chiave pubblica è quella che viene condivisa con il sistema presso il quale ci si vuole autenticare, detto relying party (RP), mentre la chiave privata è la chiave segreta che l'utente deve custodire e utilizzare per effettuare una firma al fine di dimostrare la propria identità. La generazione della coppia di chiavi avviene mediante algoritmi basati su problemi matematici non invertibili, infatti di norma viene generata la chiave privata, dalla quale viene estratta la pubblica ma non è possibile il contrario, ovvero di risalire alla chiave privata partendo da quella pubblica. Quindi l'efficacia di questo sistema risiede nella non divulgazione della chiave privata. Con la crittografia asimmetrica è possibile effettuare sia le operazioni di firma che di cifratura, invertendo chiave pubblica e privata. Infatti, per cifrare si utilizzerà la chiave pubblica e per decifrare quella privata, in questo modo solo il possessore della chiave privata avrà accesso ai dati. Per quanto riguarda la firma invece, si cifrano dei dati con la chiave privata e si decifrano con quella pubblica permettendo quindi di verificare il possesso di una determinata chiave privata complementare alla pubblica appena usata.

I due meccanismi principali che permettono il funzionamento della crittografia a pubblica per l'autenticazione presso un RP sono la registrazione e la firma.

2.2.1 Registrazione

Per registrare una nuova identità presso un RP occorre prima di tutto generare due chiavi (pubblica e privata) nuove. Dopodiché l'utente invia la propria chiave pubblica al relying party, il quale la salverà nel proprio

database. In questo modo l'utente viene verificato tramite una sfida e non attraverso la conoscenza di un segreto condiviso (la password).

2.2.2 Firma

La firma è il meccanismo che consente di verificare l'identità di un utente. Al fine di autenticare un utente, il relying party genera una sequenza casuale di dati detta challenge o sfida e la invia all'utente. L'utente, in possesso di un autenticatore, cifra questa sfida utilizzando la propria chiave privata e ne invia il risultato al relying party, il quale mediante decifratura attraverso la chiave pubblica e verifica della corrispondenza con la challenge originale, potrà confermare l'identità dell'utente.

2.3 ToothPic

ToothPic è una startup italiana con sede all'interno dell'I3P, l'incubatore del Politecnico di Torino che si occupa di cybersecurity. ToothPic ha sviluppato e brevettato una tecnologia basata sulla fingerprint specifica di ogni sensore fotografico che permette di utilizzare questa caratteristica unica del sensore per proteggere le chiavi crittografiche generate dallo smartphone. La fingerprint del sensore è l'equivalente dell'impronta digitale umana, ci permette di distinguere ogni sensore, quindi ogni dispositivo attraverso l'estrazione di un pattern di imperfezioni presenti nei pixel che compongono il sensore. Sfruttando questa tecnologia ha poi sviluppato un'applicazione per smartphone conforme allo standard FIDO2 per l'autenticazione a chiave pubblica. L'autenticatore è un'applicazione disponibile sia per Android che per iOS e permette di registrarsi sui siti abilitati al login tramite FIDO senza doversi preoccupare di generare e ricordarsi alcuna password. Le chiavi vengono infatti generate nuove per ogni nuovo relying party e ne viene offuscata la parte privata. Inoltre, per poter autorizzare un'operazione come la generazione di una nuova credenziale o la firma è solitamente richiesta la verifica da parte dell'utente mediante impronta digitale (User Verification).

2.3.1 La Tecnologia ToothPic

Come anticipato, la tecnologia di ToothPic si basa sullo sfruttamento di una particolarità dei sensori fotografici, ovvero, di alcune imperfezioni del wafer di silicio che viene utilizzato per produrre il sensore. Queste imperfezioni sono totalmente casuali e non riproducibili, per questo risulta impossibile avere due sensori con la stessa fingerprint. Questa viene quindi definita PUF (Physical Unclonable Function). Essendo questa una proprietà intrinseca del chip, a differenza dei codici inserti in fase di produzione come il MAC address nelle interfacce di rete, non può essere decisa dal produttore del sensore, quindi non può essere clonata né modificata.

Lo sfruttamento di questa particolare proprietà avviene mediante lo scatto di alcune fotografie, da queste poi viene estratta la fingerprint del sensore. Si noti che uno dei requisiti per il funzionamento della tecnologia è la possibilità da parte dello smartphone di scattare fotografie in formato RAW, quindi senza compressione, in modo da poter rendere impossibile lo sfruttamento delle immagini già presenti, ad esempio, sui social network (solitamente in formati compressi come il JPEG) per l'estrazione della fingerprint.

Come viene impiegata

La fingerprint viene estratta da un set di fotografie scattate dallo smartphone e viene ricalcolata (scattando nuove foto) ogni volta che è richiesto il suo utilizzo, ad esempio, per generare una nuova coppia di chiavi o per firmare dei dati. In particolare, l'utilità della fingerprint è quella di offuscare e de offuscare la chiave privata della coppia. Vediamo i due esempi principali di creazione di una nuova coppia di chiavi e di firma tramite chiave privata.

Una volta generata una nuova coppia di chiavi pubblica e privata, è necessario conservare segretamente la parte privata, viene quindi calcolata la fingerprint del sensore per offuscare e quindi salvare la chiave. A questo punto, l'unico modo per ottenere nuovamente accesso alla chiave privata è quello di ricalcolare la fingerprint. Per eseguire una firma, infatti, si compie questo passaggio in modo da de offuscare la componente privata ed eseguire la firma. Si noti che la chiave offuscata non rileva alcun dettaglio della chiave privata, per questo motivo per apportare un attacco sarebbe necessaria anche la fingerprint. Tuttavia, questa non viene mai salvata

sul dispositivo, ma calcolata ogni volta, rendendone quindi impossibile la sottrazione da parte di un attaccante.

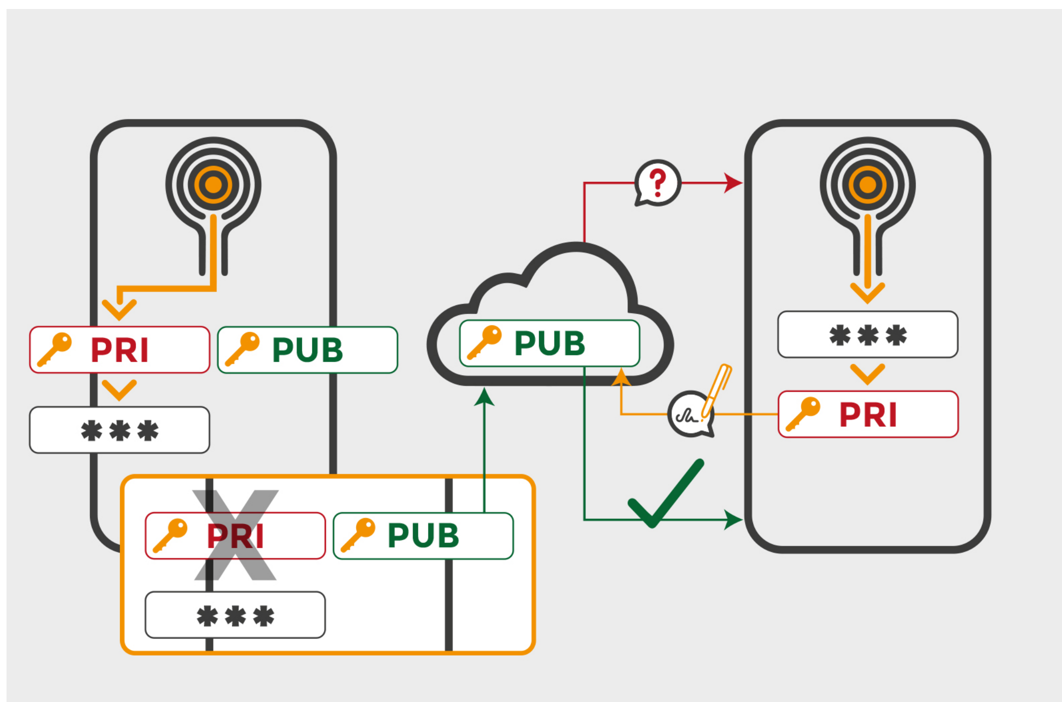

Figura 2.2. La tecnologia ToothPic.

2.3.2 L'autenticatore e la certificazione

L'autenticatore sviluppato da ToothPic [12] è conforme allo standard FIDO2 e implementa l'ultima versione disponibile dello standard, il CTAP2. L'autenticatore utilizzato durante lo sviluppo della libreria è stato sviluppato per sistema operativo Android ed è stato progettato per funzionare in due diverse modalità, tramite tecnologia ToothPic e utilizzando il keystore di Android per il salvataggio e la generazione delle chiavi. Come vedremo più avanti questo porterà a una differente risposta nella fase di creazione delle chiavi per quanto riguarda i certificati. Inoltre, da giugno 2020 l'autenticatore ha ottenuto la certificazione dalla FIDO Alliance dopo aver superato una serie di test tra cui: test di conformità per garantire

che l'implementazione fosse conforme allo standard, test di interoperabilità con server già certificati e test di sicurezza per garantire la gestione e il salvataggio sicuri delle credenziali.

Figura 2.3. Logo autenticatore certificato dalla FIDO Alliance.

2.4 OpenSSH

In questa sezione faremo una panoramica su OpenSSH e i principali tool utilizzati nello sviluppo della libreria. Open Secure Shell è una suite di programmi open source che permette la comunicazione tra computer attraverso una sessione sicura basata su protocollo SSH. Si tratta dell'alternativa con licenza open source (BSD) al software proprietario Secure Shell. OpenSSH cifra tutto il traffico in modo da prevenire attacchi quali intercettazioni, connection hijacking e altri. Inoltre, la suite di OpenSSH fornisce diversi metodi di autenticazione e programmi per la gestione delle credenziali o trasferimento di file. A partire dalla release 8.2 (14/02/2020) è stato introdotto il supporto all'autenticazione mediante autenticatori FIDO. La compatibilità viene erogata tramite una libreria di comunicazione tra OpenSSH e l'autenticatore, in pratica un'implementazione con alcune variazioni del CTAP2. Nel codice sorgente rilasciato è possibile vedere un'implementazione di questa libreria che permette la comunicazione con gli autenticatori USB, ad esempio le chiavette Yubico, o la loro versione open source le SoloKeys. Per funzionare, questa libreria, si appoggia ad un'altra libreria open source sviluppata da Yubico, libfido2, la quale si occupa della comunicazione vera e propria tra il computer e l'autenticatore e offre inoltre una linea di comando per la gestione delle chiavette

di sicurezza, come ad esempio: impostare un pin, resettare la chiavetta o controllare lo spazio disponibile per il salvataggio di nuove credenziali.

OpenSSH offre diversi metodi di autenticazione tra i quali la normale password e l'autenticazione a chiave pubblica. Fino alla versione precedente alla 8.2 occorreva generare sul proprio computer una nuova coppia di chiavi (pubblica e privata) e poi "pushare" ovvero caricare sul server a cui ci si intende connettere, la chiave pubblica. In questo modo occorre prestare attenzione a come si custodisce la chiave privata, che deve essere protetta a dovere e non deve andare persa, altrimenti non ci si potrebbe più autenticare. Va comunque segnalato che OpenSSH nella configurazione di default mantiene l'autenticazione tramite password come ultima opzione, quindi difficilmente si rischia di rimanere tagliati fuori, ma se si pensa a un ambiente di lavoro in cui ci sono vari attori con vari ruoli diversi non sempre abbiamo a disposizione la password di accesso, o questa potrebbe anche essere disattivata dagli amministratori per motivi di sicurezza. Quindi la perdita del file con la chiave privata richiederebbe l'intervento di un amministratore del sistema per ripristinare l'accesso dell'utente.

2.4.1 SSH-KEYGEN

Ssh-keygen è un tool per la creazione di chiavi di autenticazione per SSH. Per quanto riguarda le chiavi standard di SSH come ad esempio RSA o ECDSA, dopo la generazione vengono generati due file contenenti uno la parte privata e l'altro la parte pubblica della chiave. Per quanto riguarda invece le chiavi generate con autenticatori FIDO sono stati introdotti nuovi formati, come ECDSA-SK, il quale salva al suo interno un handle per identificare la chiave generata dall'autenticatore, visto che la chiave privata non viene mai esposta. Vengono inoltre salvati all'interno del file altre informazioni per l'utilizzo della chiave, come ad esempio i flag presenti in fase di creazione ("user verification" o "resident key", saranno spiegati in seguito).

2.4.2 SSH-AGENT

Ssh-agent è un tool di supporto che permette di tenere traccia le chiavi da utilizzare con SSH. Questo programma permette di utilizzare le credenziali per autenticarsi senza dover specificare il file contenente la chiave privata

(o l'handle nel nostro caso). Possiamo vedere l'agent come un portafoglio per le chiavi create.

2.4.3 SSH-ADD

Ssh-add è il tool che permette il caricamento delle chiavi generate all'interno dell'agent. È possibile eseguire il comando senza argomenti per caricare, se presenti, le chiavi con nomi standard quali:

- ~/.ssh/id_rsa
- ~/.ssh/id_dsa
- ~/.ssh/id_ecdsa
- ~/.ssh/id_ed25519
- ~/.ssh/identity

In alternativa si indica il nome del file, e il relativo path, contentente la chiave privata o l'handle.

2.5 Bluetooth Low Energy (BLE)

Per concludere questo capitolo di background, vedremo i principali concetti che caratterizzano la tecnologia Bluetooth utilizzata per la comunicazione tra l'autenticatore e OpenSSH. Il Bluetooth Low Energy (BLE) è uno standard di comunicazione wireless sviluppato dal Bluetooth Special Interest Group (Bluetooth SIG), un'organizzazione che si occupa dello sviluppo e della standardizzazione della tecnologia bluetooth. Il BLE è stato inizialmente sviluppato da Nokia nei primi anni 2000 e venne standardizzato e integrato nella versione 4.0 della tecnologia (2010), è conosciuto anche con il nome di Bluetooth Smart. Rispetto alla versione classica, il BLE è pensato per essere applicato in quegli ambiti in cui il consumo di energia è rilevante, come ad esempio l'internet of things (IoT). Il BLE è una tecnologia "always off", significa che la trasmissione dei dati avviene soltanto quando necessario, riducendo in questo modo il consumo energetico in quelle applicazioni in cui è richiesta una connessione a lungo termine ma con brevi intervalli di comunicazione, ad esempio la lettura della temperatura da un sensore.

2.5.1 Ruoli: Client e Server

Il client invia le richieste al server e ne riceve le risposte, non ha alcuna conoscenza preliminare dei servizi forniti dal server e ha come primo obiettivo quello di interrogarlo per scoprire la presenza e la natura di questi servizi attraverso la fase di “service discovery”. Una volta completata può iniziare a leggere e scrivere gli attributi del server e ricevere le notifiche.

Il server riceve le richieste e risponde, si occupa della gestione dei dati da rendere disponibili al client organizzandoli in attributi e può comunicare anche per mezzo di notifiche. Ogni dispositivo BLE deve implementare almeno un GATT server che risponda alle richieste dei client. Un semplice esempio è un sensore di temperatura.

2.5.2 UUIDs

Gli UUIDs (Universally Unique IDentifier) sono degli identificativi su 128 bit utilizzati per riconoscere i servizi offerti da un server GATT. Lo SIG fornisce una versione compatta di 16 o 32 bit degli UUID registrati per ridurre la quantità di dati trasmessi. Ad esempio, “0xFFFD” identifica un GATT service della Fido Alliance.

2.5.3 GATT, Servizi e Caratteristiche

Il GATT (Generic Attribute Profile) stabilisce come avviene lo scambio di dati attraverso una connessione BLE. Utilizza ATT (Attribute Protocol) come protocollo di trasporto per lo scambio dati tra due dispositivi. GATT definisce una gerarchia e un modello di astrazione dei dati, definendo come essi siano organizzati e scambiati tra le applicazioni. Gli attributi sono l’unità di dato su cui si basano ATT e GATT e sono organizzati in modo strettamente gerarchico: un GATT server espone uno o più servizi, i quali possono contenere zero o più caratteristiche. Esse, a loro volta, possono includere zero o più descrittori.

Servizi

I servizi compongono un GATT server e sono identificati da un UUID che permette di identificare un determinato servizio offerto dal server. Possiamo immaginare un servizio come l’equivalente di una classe in un

linguaggio di programmazione ad oggetti. Possono essere definiti servizi primari e secondari, ma i secondi sono raramente utilizzati. Come definito nello standard del CTAP2, un autenticatore basato su tecnologia BLE deve implementare il "FIDO GATT Service" identificato da "0xFFFD". I servizi sono esposti mediante il processo di advertising, ovvero la trasmissione degli UUID dei servizi forniti dal dispositivo.

Caratteristiche

Le caratteristiche sono le componenti principali di un servizio. Sono identificate da uno UUID e possiedono un campo proprietà che identifica i permessi relativi alla caratteristica, nel caso del CTAP2 sono utilizzati i seguenti: "Read", "Write", "Read/Write" e "Notify". Le caratteristiche permettono quindi di ricevere dei comandi da parte di un client e di trasmettere delle informazioni siano esse in lettura o sotto forma di notifica. Ad esempio, la caratteristica del CTAP2, "fidoStatus" è identificata da "F1D0FFF1-DEAA-ECEE-B42F-C9BA7ED623BB" ed è di tipo "Notify"; è quindi possibile, come vedremo, ricevere i dati dall'autenticatore sotto forma di notifiche dopo essersi registrati.

Capitolo 3

Progettazione della libreria BLESSH

In questo capitolo verranno analizzate le fasi della progettazione della libreria. In questo documento si farà riferimento ad essa chiamandola BLESSH (dalla composizione di BLE, Bluetooth Low Energy e SSH per indicare la suite OpenSSH) per evitare ripetizioni. Saranno inoltre presentate e discusse le varie scelte tecniche affrontate quali la scelta del linguaggio di programmazione e la selezione di librerie intermedie.

3.1 Introduzione e prerequisiti

Il software da sviluppare è una libreria condivisa che permetta la comunicazione del client OpenSSH con l'autenticatore FIDO tramite tecnologia BLE 3.1. Lo sviluppo della libreria BLESSH è avvenuta su sistema operativo Linux, in particolare Ubuntu nella versione 20.10. Questa scelta è stata fatta principalmente per la possibilità di disporre di una versione più aggiornata della suite di OpenSSH, in quanto la versione per Windows non risulta allineata a quella per sistemi Unix. Per quanto riguarda il linguaggio di programmazione invece le scelte possibili erano due: C o C++, dal momento che OpenSSH è scritto in C, la prima opzione potrebbe sembrare la più ovvia. Tuttavia, la necessità di avere una libreria condivisa con il solo vincolo di implementare tre metodi forniti da OpenSSH ha permesso di poter sfruttare il linguaggio C++ così da poter evitare, quando possibile, l'utilizzo di puntatori e allocazione dinamica e inoltre è

stato possibile sfruttare librerie esterne come, ad esempio, quella del BLE e di utilizzare uno stile di programmazione orientato agli oggetti. Oltre alle librerie necessarie che verranno discusse in seguito, è richiesta la presenza delle librerie di sviluppo di OpenSSL per sfruttare gli algoritmi di cifratura e hashing richiesti dal CTAP2.

Figura 3.1. Schema di comunicazione OpenSSH - BLESSH - FIDO.

3.1.1 I file PROTOCOL.u2f e sk-api.h

La versione di OpenSSH a cui si fa riferimento è l'ultima disponibile in questo momento (8.4). Nelle release notes, vengono indicati due file a cui prestare attenzione per un'implementazione della libreria di comunicazione, sia essa per Bluetooth o NFC, questi sono: PROTOCOL.u2f[1] e sk-api.h[2]. Il primo descrive il supporto di OpenSSH agli autenticatori FIDO e descrive con un'astrazione di alto livello il funzionamento delle procedure quali registrazione e firma e fornisce una descrizione dei campi delle strutture dati da utilizzare. Il secondo file è invece un file header (.h) scritto in linguaggio C che contiene al suo interno la definizione dell'API da implementare: contiene infatti la definizione delle strutture dati che dovranno essere popolate dalla libreria e restituite al programma chiamante (ssh-agent) e la definizione dei prototipi delle funzioni che dovranno essere implementate dalla libreria. Una descrizione dettagliata sui metodi implementati verrà data in seguito.

[1] https://github.com/openssh/openssh-portable/blob/master/PROTOCOL.u2f

[2] https://github.com/openssh/openssh-portable/blob/master/sk-api.h

3.2 La scelta delle librerie

Dopo aver fatto le scelte preliminari di sistema operativo e linguaggio di programmazione e dopo aver analizzato e studiato i file prima descritti, è iniziata la fase di progettazione vera e propria della libreria. E il primo problema affrontato è stata la scelta di una libreria (o di un framework) per la comunicazione BLE.

3.2.1 Libreria BLE

Dovendo sviluppare una libreria in C++ la soluzione più semplice sarebbe stata quella di utilizzare un framework di sviluppo che al suo interno contenesse già tutte le librerie necessarie e che fosse ben conosciuto e sviluppato: il più famoso è QT. È un framework multipiattaforma scritto in C++ disponibile sia per sistemi operativi desktop che mobile ed è disponibile sia in versione open che con licenza a pagamento. Proprio la questione della licenza ha portato a scartare questo prodotto. Infatti, la versione open source è destinata a progetti privati o scolastici, e non per utilizzi commerciali, per questo motivo confrontandomi con il tutor aziendale abbiamo concordato sulla ricerca di una soluzione alternativa.

Dopo aver effettuato diverse ricerche la scelta si è ristretta a tre librerie: GattLib++ [7], libble++ [9] e TinyB [11].

La prima è un wrap dell'omonima libreria sviluppata in C (GattLib)e permette di sfruttare le potenzialità fornite dal linguaggio C++ come le callback e le funzioni lambda, tuttavia presenta due problemi: il primo, ma anche il minore dei due è la necessita di dover installare sia questa libreria che la versione in C su cui è basata, rendendo quindi possibili problemi di incompatibilità in caso di aggiornamenti della seconda; il secondo, e anche il più grave, la mancanza del supporto all'advertising, rendendo quindi impossibile distinguere un autenticatore che presenta il servizio FIDO da un normale dispositivo bluetooth che non lo implementa.

Libble++ è una libreria sviluppata in C++ con la particolarità di non essere basata su Bluez [1] (lo stack protocollare Bluetooth ufficiale di Linux) ma su hcitool [8] (un tool per la configurazione delle connessioni Bleutooth). Il problema riscontrato con questa libreria è l'impossibilità di effettuare la scansione di dispositivi BLE se non tramite“sudo”, rendendo di fatto inutilizzabile una libreria che per essere utilizzata richieda permessi elevati.

Arriviamo dunque a TinyB, è una libreria sviluppata da Intel che espone le BLE GATT API per C++ e Java, e sfrutta lo stack protocollare di Bluez. Nonostante il suo sviluppo sia fermo dal 2017 risulta comunque utilizzabile anche se con qualche mancanza come vedremo più avanti. Questa libreria espone pochi semplici metodi che in realtà sono poi dei wrap delle funzioni di Bluez. Dopo aver compilato ed eseguito i file di esempio ho deciso di utilizzare questa libreria in quanto non poneva limitazioni critiche come le precedenti. Una volta compilata viene resa disponibile all'interno del progetto in forma di libreria condivisa.

3.2.2 Libreria CBOR

Il CTAP2 utilizza il CBOR [2] (Concise Binary Object Representation) come formato di rappresentazione dei dati per la comunicazione con l'autenticatore, il CBOR è ispirato al JSON ma è orientato alla rappresentazione di dati binari che altrimenti sarebbero rappresentati in base64 con conseguente aumento di spazio e complessità. Questo formato di rappresentazione dei dati è stato pensato principalmente per l'ambito IoT con l'obiettivo di risparmiare spazio e potenza di calcolo, è uno standard (RFC 8949) ed è implementato sotto forma di libreria per la maggior parte dei linguaggi di programmazione. Dopo aver testato varie opzioni disponibili per C/C++ ho scelto di utilizzare cborg [3] in quanto era l'unica tra quelle testate che permetteva di utilizzare un indice numerico come identificativo all'interno di una mappa. Questo vincolo deriva dallo standard del CTAP2 che, come vedremo, richiede di utilizzare indici numerici all'interno delle mappe che incorporano i dati delle richieste. La libreria utilizzata viene resa disponibile all'interno del progetto in forma di libreria statica.

3.3 BLESSH e progetto per testing

Prima di iniziare lo sviluppo vero e proprio della libreria BLESSH, ho creato un progetto di test, dotato di main così da poter evitare di dover utilizzare OpenSSH per l'esecuzione del middleware. Questo progetto rimane parallelo alla libreria vera e propria e ne condivide le stesse classi, con la differenza che una volta compilato produce un file eseguibile. Ho utilizzato questo progetto per avere la possibilità di fare un debug più semplice utilizzando l'IDE.

Capitolo 4

Sviluppo della libreria BLESSH

In questo capitolo verrà analizzato lo sviluppo della libreria, verranno descritte le funzioni implementate e richieste da OpenSSH, le strutture dati utilizzate e le classi sviluppate per il necessario funzionamento.

4.1 Strutture dati e prototipi da sk-api.h

Come anticipato, il file sk-api.h fornisce l'interfaccia della libreria dichiarando al suo interno le strutture dati necessarie per il passaggio dei dati dal middleware a OpenSSH e i prototipi delle funzioni da implementare. Si ricorda che la versione qui analizzata corrisponde alla 8.4 di OpenSSH. Per mantenere una struttura ordinata della sezione, strutture e prototipi saranno analizzati nell'ordine in cui si trovano all'interno del file header. I prototipi sono presentati con una descrizione ad alto livello, per l'approfondimento sull'implementazione si rimanda alla sezione successiva. Per evitare ripetizioni, si anticipa che, tutte le strutture o i parametri con il suffisso "_len" all'interno del file header indicano la lunghezza del relativo dato; si tratta di vettori di bytes o vettori di strutture. Si noti inoltre che, escludendo la funzione sk_api_version, il valore di ritorno delle altre funzioni implementate è un intero che indica il successo dell'operazione (valore 0) o uno degli errori predefiniti di OpenSSH (valori diversi da 0).

4.1.1 Struttura sk_enroll_response

Questa struttura dati viene utilizzata per incapsulare i dati provenienti dall'autenticatore in seguito a una richiesta di generazione di una nuova credenziale. Vediamo i campi e il formato richiesto:

- public_key: vettore di bytes contenente i dati relativi alla chiave pubblica generata per un determinato utente e un determinato RP. Si tratta di una coordinata (X,Y) della curva ellittica, X e Y sono a loro volta dei vettori di bytes. Per rappresentarla, si usa la notazione "octet-string" [17]. Il risultato sarà nella forma $\{0x04\|X\|Y\}$.

- key_handle: vettore di bytes contente l'identificativo della credenziale generata dall'autenticatore necessario per identificare la chiave da utilizzare nella fase di firma.

- signature: vettore di bytes contenente la challenge firmata.

- attestation_cert: vettore di bytes contenente i dati relativi ai certificati. Nel caso dell'autenticatore ToothPic si tratta di un self-signed, oppure (se si utilizza il keystore) una catena di certificati comprendenti quello del produttore dello smartphone e quello di Google. Il certificato, o la catena, servono a garantire la provenienza dei dati, infatti la chiave pubblica viene firmata con la privata di attestazione associata al modello di smartphone. OpenSSH se volesse, potrebbe quindi decidere (in futuro) di verificare la catena dei certificati e verificare la provenienza degli autenticatori [14].

- authdata: vettore di bytes contenente la struttura dati "authenticator data" [1]. Dalla figura 4.1 possiamo vedere che contiene i seguenti campi di nostro interesse:

  - RP ID hash: SHA-256 dell' RP ID a cui corrisponde la credenziale.
  - FLAGS: i flags settati dall'autenticatore in fase di generazione delle credenziali, tra i quali ad esempio "UV" e "UP" per segnalare user verification e presence.

[1] https://www.w3.org/TR/webauthn/#authenticator-data

– COUNTER: un intero progressivo che viene incrementato a ogni operazione di firma andata a buon fine. Utilizzato per identificare la clonazione di un autenticatore.
– CREDENTIAL ID: il key_handle per identificare la credenziale.
– CREDENTIAL PUBLIC KEY: struttura dati contenente la chiave pubblica.

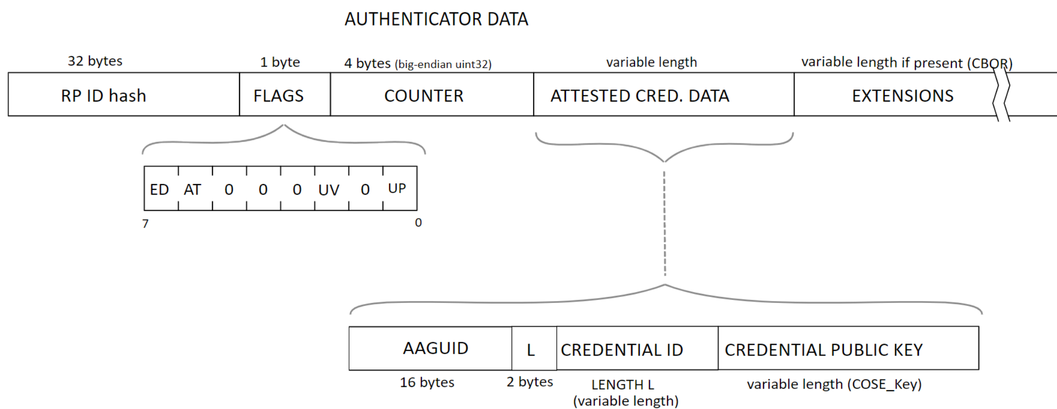

Figura 4.1. Composizione della struttra dati Authenticator Data

4.1.2 Struttura sk_sign_response

All'interno di questa struct sono contenuti i dati generati dall'autenticatore in seguito ad un'operazione di firma.

- flags: byte contente i flag riportati dall'autenticatore all'interno della struttura authenticator data
- counter: un intero progressivo che viene incrementato a ogni operazione di firma andata a buon fine. Utilizzato per identificare la clonazione di un autenticatore.
- sig_r e sig_s: due vettori di bytes contenenti le componenti R e S di una firma ECDSA [13]. Dall'immagine possiamo vedere ad alto livello come avviene la firma dei dati da parte dell'autenticatore, clientDataHash equivale alla challenge.

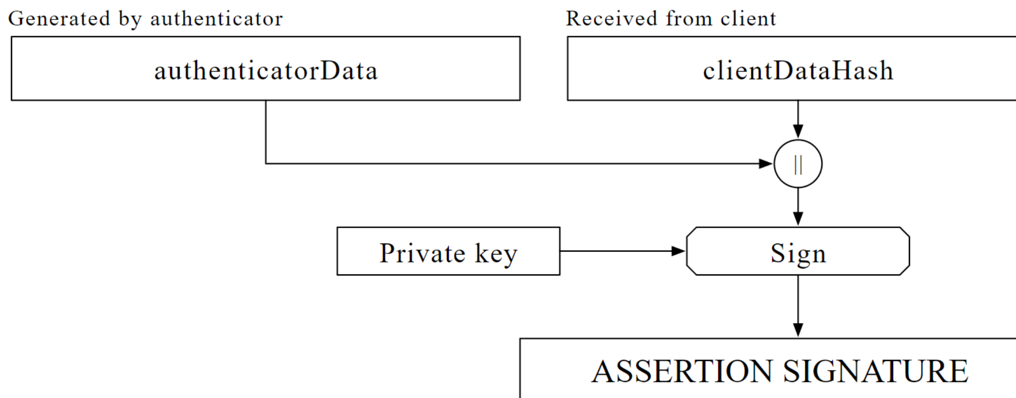

Figura 4.2. Operazione di firma da parte dell'autenticatore

4.1.3 Struttura sk_resident_key

Struttura dati contenente le informazioni utili per identificare una resident key. Si parla di "resident key" quando la credenziale rimane salvata all'interno dell'autenticatore. Si noti che l'autenticatore ToothPic lavora solo con resident keys.

- alg: intero atto a identificare l'algoritmo utilizzato e quindi il tipo di credenziale (in questo caso sempre 0x00 ovvero ECDSA).
- slot: non utilizzato.
- application: stringa contenente l'identificativo dell'RP ("ssh:").
- key: si tratta di una struttura dati del tipo sk_enroll_response. (Non tutti i campi saranno riempiti come vedremo).
- flags: byte contenente i flag utilizzati nella fase di generazione della credenziale.

4.1.4 Struttura sk_option

Struttura a tre campi per identificare delle opzioni che possono essere passate da linea di comando a OpenSSH.

- name: stringa (vettore di caratteri terminato da '0') per identificare il nome dell'opzione.

- value: stringa (vettore di caratteri terminato da '0') per identificare il valore dell'opzione.
- required: intero che indica se l'opzione deve essere supportata dalla libreria.

4.1.5 Prototipo sk_api_version

```
#define SSH_SK_VERSION_MAJOR 0x00070000

uint32_t sk_api_version(void);
```

Questa funzione deve soltanto ritornare l'intero definito come costante SSH_SK_VERSION_MAJOR all'interno del file e serve a identificare la versione della libreria. Questo valore cambia con il variare delle versioni di OpenSSH nel caso queste apportino modifiche alle strutture dati o ai prototipi. Nel caso analizzato il valore è: 0x00070000.

4.1.6 Prototipo sk_enroll

```
int sk_enroll(uint32_t alg, const uint8_t *challenge,
        size_t challenge_len, const char *application,
        uint8_t flags, const char *pin,
        struct sk_option **options,
        struct sk_enroll_response **enroll_response);
```

Il prototipo relativo alla funzione che si occupa di generare una nuova credenziale, ha come corrispettivo lato CTAP2 il metodo "authenticatorMakeCredential" [2]. Questa funzione si occupa di generare e trasmettere la richiesta all'autenticatore e di elaborarne la risposta popolando i campi della struttura sk_enroll_response. Questa è la funzione che viene invocata dal tool ssh-keygen quando si vuole generare una nuova credenziale.

- alg: algoritmo da utilizzare (ECDSA).

[2] https://fidoalliance.org/specs/fido-v2.0-ps-20190130/fido-client-to-authenticator-protocol-v2.0-ps-20190130.html#authenticatorMakeCredential

- challenge: vettore di bytes contenente una sfida da firmare dopo aver generato le credenziali.
- application: stringa contenente l'identificativo dell'RP ("ssh:").
- flags: byte contenente i flag da passare all'autenticatore (es:"user verification" o "resident key").
- pin: pin inserito dall'utente da utilizzare come meccanismo di "user verification" nell'autenticatore.
- options: vettore delle opzioni passate da linea di comando.
- enroll_response: puntatore alla struttura dati da allocare e popolare con i dati provenienti dall'autenticatore.

4.1.7 Prototipo sk_sign

```
int sk_sign(uint32_t alg, const uint8_t *message,
        size_t message_len, const char *application,
        const uint8_t *key_handle,
        size_t key_handle_len,
        uint8_t flags, const char *pin,
        struct sk_option **options,
        struct sk_sign_response **sign_response);
```

Questo è invece il prototipo della funzione che si occupa di effettuare una firma con una data credenziale, corrisponde al metodo "authenticator-GetAssertion" [3] nel CTAP2 e di fatto è la funzione che permette di autenticarsi su un server OpenSSH utilizzando l'autenticatore. Questa funzione viene invocata dal tool ssh nel caso venga indicato esplicitamente quale chiave usare per l'autenticazione. Altrimenti viene invocata da ssh-agent il quale si occupa di trovare quale chiave in suo possesso è compatibile con la chiave pubblica presente sul server.

- alg: algoritmo da utilizzare (ECDSA).

[3] https://fidoalliance.org/specs/fido-v2.0-ps-20190130/fido-client-to-authenticator-protocol-v2.0-ps-20190130.html#authenticatorGetAssertion

- message: vettore di bytes contenente i dati da firmare, si tratta della challenge.
- application: stringa contenente l'identificativo dell'RP ("ssh:").
- key_handle: vettore di bytes contenente l'identificativo della chiave da utilizzare per la firma.
- flags: byte contenente i flag da passare all'autenticatore (es: "user verification" o "resident key").
- pin: pin inserito dall'utente da utilizzare come meccanismo di "user verification" nell'autenticatore.
- options: vettore delle opzioni passate da linea di comando.
- sign_response: puntatore alla struttura dati da allocare e popolare con i dati provenienti dall'autenticatore.

4.1.8 Prototipo sk_load_resident_keys

```
int sk_load_resident_keys(const char *pin,
        struct sk_option **options,
        struct sk_resident_key ***rks, size_t *nrks);
```

L'ultimo prototipo necessita di alcune puntualizzazioni rispetto ai precedenti, si tratta di una funzione il cui scopo è quello di "effettuare il download delle resident keys dall'autenticatore". Come già detto, però, le chiavi private non vengono mai esposte dall'autenticatore e per questo la denominazione di questo prototipo potrebbe sembrare fuorviante. In realtà si tratta di fare il download degli identificativi delle resident keys presenti nell'autenticatore, allo stesso modo in cui preleviamo handle e chiave pubblica nella sk_enroll. Questa funzione non ha un metodo corrispondente nel CTAP2 implementato dall'autenticatore ma esiste nella versione 2.1 [16] del protocollo che tuttavia non è ancora standard ma in bozza. Si tratta di una sotto parte del metodo "authenticatorCredentialManagement" [4] e ne ha richiesto un'implementazione basica lato

[4] https://fidoalliance.org/specs/fido2/fido-client-to-authenticator-protocol-v2.1-rd-20191217.html#authenticatorCredentialManagement

autenticatore per garantire il funzionamento con OpenSSH senza tuttavia implementare tutta la nuova versione del CTAP 2.1. Questa funzione può essere invocata da due differenti tool: ssh-add e ssh-keygen. Vedremo verso la fine le differenze e degli esempi di utilizzo.

- pin: pin inserito dall'utente da utilizzare come meccanismo di "user verification" nell'autenticatore.

- options: vettore delle opzioni passate da linea di comando.

- rks: puntatore a un vettore di sk_resident_key da allocare e popolare con i dati degli handle scaricati dall'autenticatore.

- nrks: numero di handle di resident keys scaricati.

4.2 Implementazione dei metodi della libreria

In questa sezione saranno analizzate le implementazioni dei metodi precedentemente descritti in forma di prototipo e verranno presentate le classi ausiliarie sviluppate per questo progetto. Per questioni di ordine, le classi ausiliarie verranno descritte all'interno del blocco in cui saranno presentate per la prima volta. La descrizione delle funzionalità è fatta sulla base di un diagramma di flusso in modo da semplificare e riassumere i vari passaggi. Si noti che all'interno dei diagrammi ci sono dei blocchi che sono identici in una o più funzioni (ad esempio la parte di connessione all'autenticatore) e pertanto non saranno ripetuti nelle successive descrizioni.

4.2.1 Funzione sk_api_version

Come anticipato questa funzione si occupa soltanto di ritornare un valore intero contente la versione del file sk-api.h su cui è basato il middleware. Nel nostro caso 0x00070000, il valore settato da OpenSSH nella versione 8.4. Questo permette di rilevare un eventuale incompatibilità tra la versione su cui è basta la libreria e la versione installata di OpenSSH.

4.2.2 Funzione sk_enroll

Approfondiamo ora una delle tre maggiori funzioni da implementare, questa ci permette di generare una nuova coppia di chiavi tramite l'autenticatore e restituirne la parte pubblica a OpenSSH. Con l'ausilio del flow chart 4.3, analizziamo i blocchi principali che compongono la funzione della libreria.

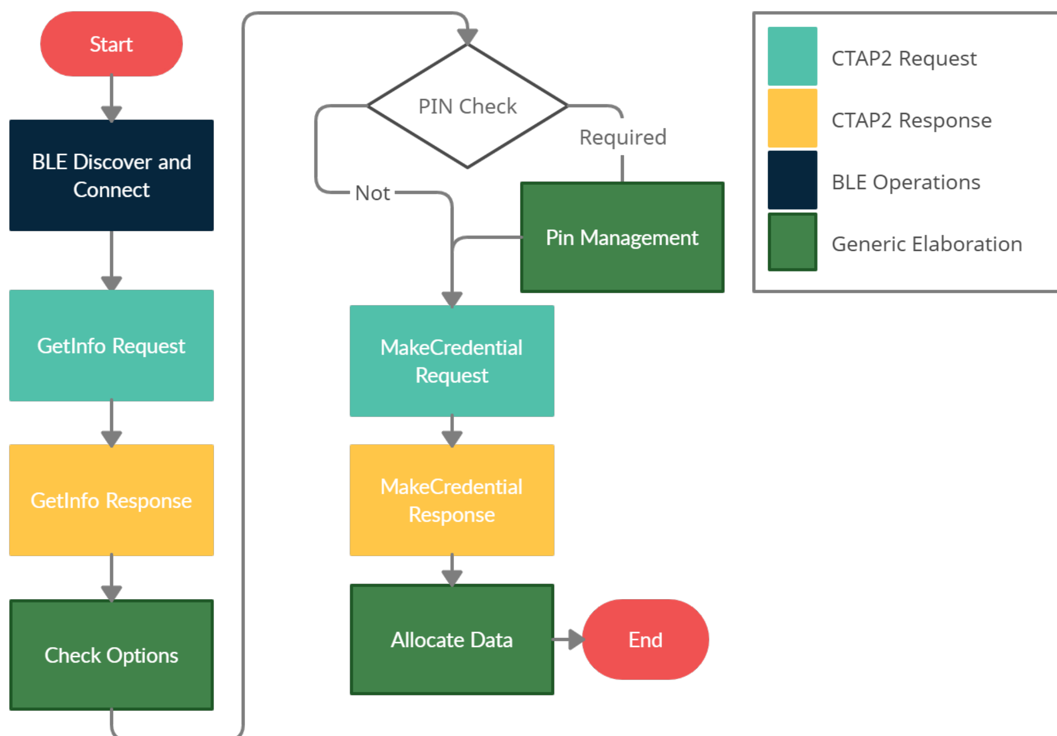

Figura 4.3. Diagramma del flusso di esecuzione della funzione di enroll.

BLE Discover and Connect

Questa sezione della funzione fa uso delle classi ausiliarie BLEManager e AuthenticatorService. La prima viene utilizzata per avviare il processo di "discovery" e quindi trovare un dispositivo BLE che esponga il servizio "FIDO GATT Service". La seconda classe invece viene utilizzata per inviare all'autenticatore la richiesta precedentemente creata.

La classe BLEManager

La classe BLEManager ha il compito di fare da wrapper alla libreria tinyB. È progettata seguendo un pattern singleton, che permette di istanziare la classe una sola volta. Questa classe espone pubblicamente le quattro caratteristiche definite dal CTAP2. Inoltre, espone quattro metodi per interagire con l'autenticatore:

- discoverAndConnect: si occupa di avviare il processo di discovery di dispositivi bluetooth e si connette a quello che espone il servizio "FIDO GATT Service". Una volta connesso procede a salvare tra i membri della classe le caratteristiche del CTAP2.

- disconnect: effettua la disconnessione dall'autenticatore e libera le risorse ad esso associate.

- readCharacterstic: questo metodo permette di leggere direttamente una caratteristica, senza usufruire del meccanismo delle notifiche. Viene utilizzato per la lettura delle caratteristiche "fidoControlPointLength" in modo da ottenere il valore della MTU da utilizzare, e "fidoServiceRevisionBitfield" per ottenere le versioni del protocollo CTAP implementate dall'autenticatore.

- sendMessage: come nelle reti IP anche nella comunicazione bluetooth esiste il concetto di MTU (Maximum Transmission Unit) la dimensione massima che può avere un pacchetto da trasmettere via BLE. Il metodo si occupa quindi, prima d'inviare i dati, di scomporre il messaggio in frames seguendo quanto descritto dal CTAP2[5] e poi procede con la scrittura dei valori nella caratteristica ricevuta come parametro.

La classe AuthenticatorService

La classe AuthenticatorService ha il compito di ricomporre i messaggi ricevuti dall'autenticatore e di farne un'analisi iniziale. Si occupa infatti di concatenare e salvare in un unico vettore i vari frames ricevuti e inoltre tiene traccia del codice di stato ricevuto così da poter riconoscere subito un errore prima di procedere con la parsificazione dei dati. Per via della

[5] https://fidoalliance.org/specs/fido-v2.0-ps-20190130/fido-client-to-authenticator-protocol-v2.0-ps-20190130.html#ble-framing-fragmentation

natura asincrona del BLE è stato implementato un meccanismo di condition variable per l'estrazione della risposta da questa classe. Il chiamante si troverà quindi in attesa della ricezione di tutta la comunicazione prima di poter accedere ai dati. Questo meccanismo è basato sulla definizione dei messaggi CTAP2 in quanto all'inizio del primo frame è indicata la lunghezza totale del messaggio trasmesso. Tuttavia, la CV si sblocca anche in caso di timeout evitando quindi un'attesa infinita. Il discorso del timeout verrà ripreso nelle conclusioni in quanto presenta delle criticità.

GetInfo Request

Come indicato dalle linee guida del CTAP2, la prima operazione da eseguire in una comunicazione con l'autenticatore è la "authenticatorGetInfo" [6] si tratta di una richiesta da fare all'autenticatore con il fine di conoscere da quest'ultimo quali sono le opzioni, i protocolli e le estensioni supportate. OpenSSH, come possiamo vedere dalla libreria per la comunicazione via USB, in realtà non si preoccupa di eseguire questa richiesta ma procede direttamente con quelle successive andando a gestire eventuali errori, tuttavia, è stato scelto di procedere in questo modo per uniformarsi a quanto indicato dallo standard. In questa fase, la libreria genera la richiesta in un vettore di bytes. Per questa richiesta non è stato fatto uso di una classe ausiliaria in quanto si tratta di una richiesta costante senza parametri.

GetInfoResponse

Attraverso il sistema delle notifiche, e sfruttando nuovamente la classe AuthenticatorService viene elaborata la risposta dell'autenticatore e i dati sono analizzati e salvati in un'istanza della classe GetInfoResponse. Questa classe ausiliaria si occupa di decodificare i dati ricevuti in forma CBOR e salvare le opzioni come valori booleani (es: “user verification” o “resident key”). Come altre classi ausiliarie che manipolano i dati in formato CBOR si affida alla libreria cborg descritta in precedenza.

[6] https://fidoalliance.org/specs/fido2/fido-client-to-authenticator-protocol-v2.1-rd-20191217.html#authenticatorGetInfo

Check Options + Pin Check

Questa fase serve innanzitutto a capire se l'autenticatore supporta tutte le opzioni che vengono richieste da OpenSSH, ad esempio: "user verification" o "resident key". La prima opzione è un livello di sicurezza ulteriore definito nel CTAP2, si tratta di verificare l'identità dell'utente con metodi sicuri quali la biometria. Viene inoltre verificato se l'autenticatore supporta e richiede l'immissione di un PIN come metodo di verifica più debole. Se il pin è supportato e richiesto la funzione sk_enroll ritorna un errore predefinito (-3) il quale porterà OpenSSH a rieseguire la stessa funzione da capo ma con una richiesta di inserimento del PIN da parte dell'utente.

Pin Management

Nel caso in cui fosse richiesto il PIN in questa fase occorre effettuare tutte le operazioni necessarie ad utilizzare il PIN come specificato dal protocollo CTAP2[7]. Attraverso la classe ausiliaria PinManager si gestiscono tutte le rischieste, le risposte e l'elaborazione dei dati necessari a generare un PinToken. Si tratta di un meccanismo di sicurezza volto a evitare l'invio del PIN impostato, infatti non sono inviati né il PIN e nemmeno una forma cifrata, ma un "token" ottenuto dall'autenticatore rigenerato ad ogni avvio. Il pinToken è scambiato in forma cifrata utilizzando come chiave un segreto condiviso che a sua volta viene generato mediante ECDH (Diffie-Hellman a Curva Ellittica). Per ottenere il pinToken è necessaria la conoscenza del PIN, in quanto ne vengono cifrati i primi 16 bytes del suo hash attraverso il segreto condiviso, questo permette all'autenticatore di verificare l'effettiva conoscenza del pin da parte dell'utente. La classe ausiliaria PinManager ha due compiti principali: la creazione delle richieste relative al meccanismo del pinToken e l'implementazione di funzioni wrapper per le funzioni crittografiche di OpenSSL come, ad esempio, la generazione dell'HMAC o il calcolo dell'hash e la sua cifratura/decifratura.

[7] https://fidoalliance.org/specs/fido-v2.0-ps-20190130/fido-client-to-authenticator-protocol-v2.0-ps-20190130.html#authenticatorClientPIN

MakeCredential Request

Una volta ottenuto (se necessario) il pinToken si procede con la creazione della richiesta per la generazione di una nuova credenziale sull'autenticatore. Attraverso la classe MakeCredential, si genera una richiesta di authenticatorMakeCredential da inviare all'autenticatore. L'invio della richiesta avviene nuovamente mediante la classe BLEManager che si occupa di gestire la comunicazione. I principali parametri che compongono una richiesta di generazione delle credenziali sono: la challenge (dati casuali inviati da firmare in seguito alla generazione delle chiavi), il RP (come sempre nel nostro caso "ssh:"), i dati relativi all'utente (specificabile da linea di comando), il tipo di chiave da generare e l'algoritmo da utilizzare ("public-key" e ECDSA) ed eventualmente i dati relativi al pinToken. La classe ausiliaria MakeCredential espone soltanto il costruttore e un attributo (il vettore di bytes contenente la richiesta in formato CBOR). Il costruttore riceve tutti i parametri prima descritti e si occupa di generare la richiesta in formato CBOR, infine, attraverso una funzione di utility aggiunge all'inizio della richiesta l'header richiesto dal CTAP2[8].

MakeCredential Response

Attraverso l'utilizzo della classe ausiliaria AuthenticatorService vengono salvati i dati della risposta ricevuta dall'autenticatore mentre si sfrutta la classe MakeCredentialResponse per analizzare e parsificare i dati utili a popolare la struttura sk_enroll_response. Una volta ricevuti tutti i dati dall'autenticatore si procede con la disconnessione dallo stesso. La classe ausiliaria MakeCredentialResponse ha il compito di decodificare le informazioni ricevute in formato CBOR e di estrarre i dati che saranno poi necessari a OpenSSH, come ad esempio la firma della challenge o l'identificativo della chiave generata. Inoltre si occupa di convertire la chiave pubblica dal formato fornito dall'autenticatore X,Y nel formato richiesto da OpenSSH {0x04‖X‖Y}. La classe inoltre espone un valore booleano per identificare la corretta estrazione dei dati dal vettore di ingresso. In questo modo è possibile da parte del chiamante controllare l'esito della decodifica dei dati.

[8] https://fidoalliance.org/specs/fido-v2.0-ps-20190130/fido-client-to-authenticator-protocol-v2.0-ps-20190130.html#ble-framing

Allocate Data

Come anticipato, una volta effettuata la disconnessione dall'autenticatore si procede con l'allocazione e la popolazione della struttura dati che permette a OpenSSH di sfruttare i dati ricevuti dall'autenticatore. In questa fase viene allocato dinamicamente il puntatore ricevuto come parametro, sarà poi compito del chiamante (OpenSSH) rilasciare la memoria al fine di evitare dei leak.

4.2.3 Funzione sk_sign

Questa funzione ha lo scopo di ottenere dall'autenticatore la firma di un messaggio utilizzando una particolare chiave, nella pratica è la funzione che permette di autenticarsi su un server SSH. Sempre con l'ausilio del flow chart 4.4 analizziamo i principali step da compiere.

Come possiamo notare, la prima parte della funzione è sostanzialmente identica alla sk_enroll, infatti come sempre occorre ottenere le informazioni sull'autenticatore e controllare che tutte le opzioni ricevute da OpenSSH siano supportate. Dopodiché si verifica se sia necessario l'utilizzo del pin e quindi eventualmente si avvia la procedura per l'ottenimento del pinToken.

GetAssertion Request

Per generare la richiesta di firma da parte del dispositivo di autenticazione, si utilizza la classe ausiliaria GetAssertion per formare una richiesta di tipo authenticatorGetAssertion. I principali parametri di questa richiesta sono: il messaggio da firmare, il RP (come sempre "ssh:"), l'identificativo della chiave da utilizzare in fase di firma, le varie opzioni definite da OpenSSH e eventualmente il pinToken se richiesto dall'autenticatore. Attraverso la classe BLEManager poi, si invia la richiesta all'autenticatore e si attende la risposta sotto forma di notifica. La classe ausiliaria GetAssertion è sviluppata in modo simile alla precedentemente descritta MakeCredential, si occupa quindi di esporre costruttore e vettore di bytes contenente il corpo della richiesta. La differenza chiaramente si trova nei parametri richiesti dal CTAP2.

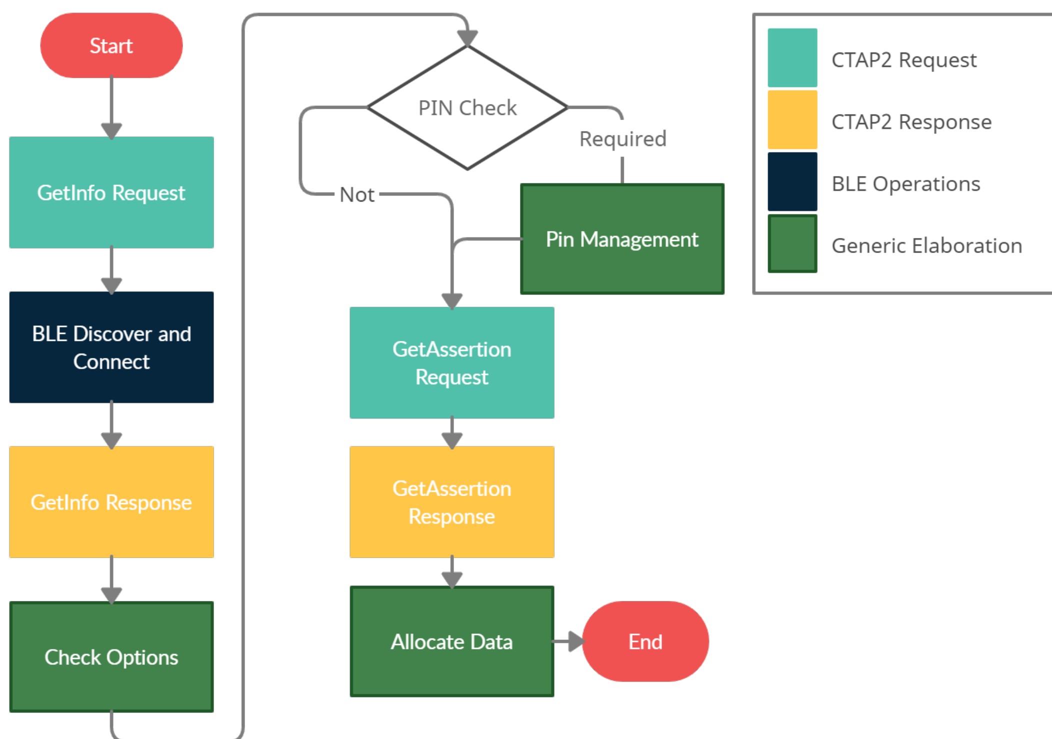

Figura 4.4. Diagramma del flusso di esecuzione della funzione di sign.

GetAssertion Response

Una volta ricevuti tutti i dati, viene effettuata la disconnessione dall'autenticatore per poi procedere con la parsificazione attraverso la classe ausiliaria GetAssertionResponse. Questa ha il compito di decodificare e parsificare i dati ricevuti. Inoltre, dispone di un metodo privato per estrarre le componenti R e S dalla firma, questo metodo si basa sulle funzioni della libreria OpenSSL.

Allocate Data

Per concludere, la funzione alloca lo spazio in memoria necessario alla struttura sk_sign_response e ne popola i campi con i dati provenienti dalla classe precedentemente descritta. Anche in questo caso sarà compito del chiamante procedere alla liberazione della memoria occupata.

4.2.4 Funzione sk_load_resident_keys

L'ultima delle funzioni implementate è quella che si occupa di fare il download dei dati relativi agli handle delle resident keys salvate sull'autenticatore. Anche in questo caso la prima operazione da fare è relativa alle informazioni sull'autenticatore effettuando una "authenticatorGetInfo". Questa volta però dovremo assicurarci che nella risposta ricevuta il flag "rk" sia presente e impostato su "true" altrimenti vorrà dire che l'autenticatore non supporta le resident keys. Inoltre, come si vede dal diagramma 4.5, non viene fatto il controllo sul pin in quanto (per decisione di OpenSSH) questa funzione richiede sempre l'utilizzo del pin come meccanismo di sicurezza.

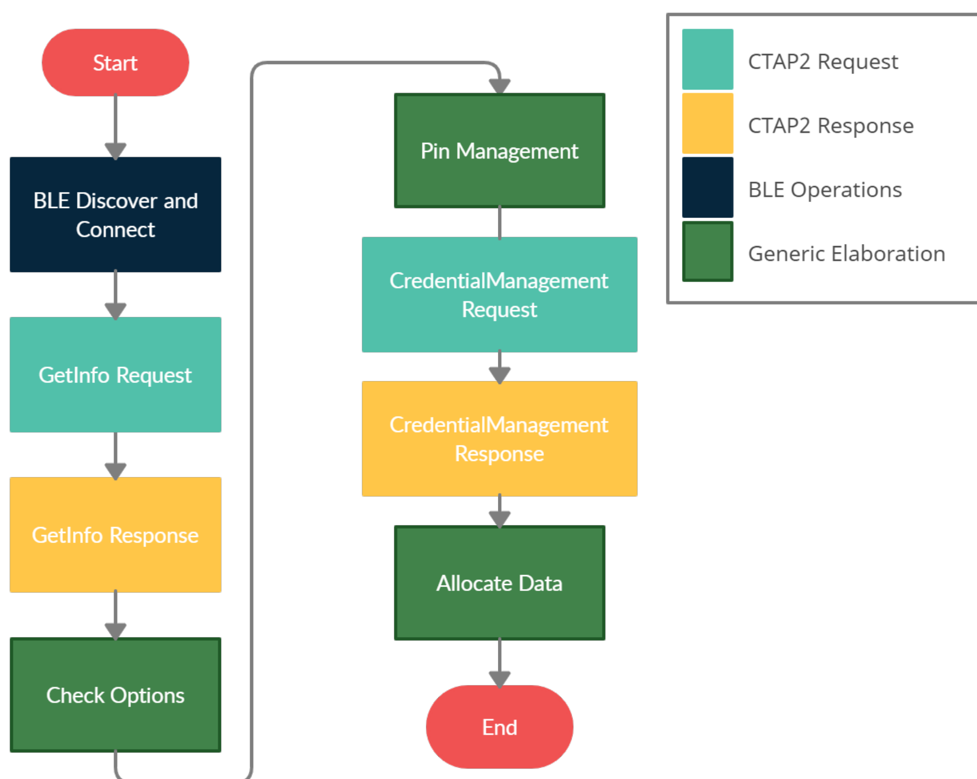

Figura 4.5. Diagramma del flusso di esecuzione della funzione di load.

Modifiche lato autenticatore

Prima di vedere la descrizione della funzione dal lato della libreria occorre specificare il funzionamento da parte dell'autenticatore. Come anticipato questa funzionalità sarà introdotta nello standard CTAP 2.1 che al momento si trova ancora in bozza. Va segnalato che nella definizione del CTAP 2.1 del metodo authenticatorCredentialManagement sono definiti diversi sotto comandi tra i quali: enumerateCredentialsBegin e enumerateCredentialsGetNextCredential. Questi due sono quelli utili al funzionamento della libreria, pertanto sono stati implementati unicamente per garantire il funzionamento della funzione richiesta da OpenSSH. Si tratta di due comandi separati, attraverso il primo si ottiene l'handle della prima resident key e il numero totale di resident keys generate per un dato RP. Il secondo va utilizzato in seguito, all'interno di un ciclo di richieste, per ottenere le rimanenti informazioni. Si tratta quindi di un comando stateful a differenza dei comandi stateless sfruttati fino ad ora. Le modifiche necessarie all'autenticatore per implementare questa nuova funzionalità si sono limitate alla creazione di una nuova classe per gestire la richiesta e la risposta, e l'inserimento di un nuovo ramo di scelta alla ricezione di un comando, in modo da poter chiamare il metodo per la gestione delle credenziali salvate. Secondo la definizione del CTAP 2.1, il supporto alle funzionalità di credential mangement (oltre a quelle appena elencate troviamo anche la possibilità di enumerare tutti i relying party o la cancellazione di credenziali) va indicato all'interno della risposta all'authenticatorGetInfo dell'autenticatore settando il flag come avviene per "user verification" o "resident key". Tuttavia, è stato deciso di non inserire all'interno della risposta l'informazione sulla presenza della nuova funzionalità in quanto non implementata interamente poiché questo avrebbe potuto portare problemi con l'utilizzo dell'autenticatore con eventuali server già basati sul CTAP 2.1.

Ai fini della gestione del comando stateful si è fatto affidamento sull'utilizzo delle sharedPreferences di Android per salvare le informazioni necessarie a inviare i dati delle resident keys successive alla prima. Si tratta dei seguenti dati: l'hash dell'RP in quanto viene inviato all'autenticatore soltanto alla prima richiesta, l'indice della prossima resident key da inviare e lo stato di validità del comando. Infatti, occorre fare in modo che non possano essere inviate le credenziali successive se non è prima arrivato il comando di begin.

CredentialManagement Request

Come anticipato, ci possono essere due tipi di richieste per il download delle resident keys, attraverso la classe CredentialManagement la libreria gestisce la creazione di entrambi i tipi di richiesta. La distinzione avviene con un parametro booleano nel costruttore. Le principali differenze tra i due comandi sono le seguenti: il codice del sotto comando ovviamente, la presenza dell'hash dell'RP nella richiesta iniziale e la presenza del pinToken nella richiesta iniziale.

CredentialManagement Response

Come per la richiesta, anche la risposta si presenta in due formati differenti a seconda del tipo di richiesta effettuata. La gestione avviene mediante la classe ausiliaria CredentialManagementResponse che si occupa di decodificare i dati in formato CBOR. Per distinguere il tipo di risposta si fa nuovamente uso di un parametro aggiuntivo nel costruttore. Anche questa classe come la MakeCredentialResponse ha al suo interno una funzione privata per la trasformazione della chiave pubblica nel formato richiesto da OpenSSH.

Allocate Data

Per finire anche questa funzione ha la necessità di allocare la memoria necessaria per i risultati ottenuti. In particolare, in questo caso occorre allocare un vettore di strutture sk_resident_key. Si è preferito (per rimanere in linea con la versione della libreria per USB) non allocare subito il vettore con la dimensione indicata dal numero di resident keys ottenuto dall'autenticatore, ma procedere a una riallocazione progressiva alla ricezione dei nuovi dati. In questo modo l'allocazione avviene soltanto per gli effettivi handle scaricati, evitando il caso in cui un autenticatore ritorni un numero sbagliato (molto grande) di chiavi costringendoci a uno spreco di memoria se non addirittura a problemi di saturazione.

4.3 Testing della libreria

Per la verifica del funzionamento della libreria ho fatto affidamento principalmente a OpenSSH nella versione 8.4 e al progetto di test precedentemente anticipato. Per quanto riguarda l'autenticatore, è stato utilizzato quello certificato di ToothPic basato su Android. Data la natura del progetto (libreria a caricamento dinamico) non ho avuto modo di testare la libreria in sé in quanto avrei necessitato di dover scrivere un'applicazione che caricasse la libreria per poi sfruttarne i metodi esposti. Ho allora deciso di sfruttare il progetto di test in modo da poter verificare il funzionamento dell'invio e della ricezione dei dati e della loro codifica. Mentre, per quanto riguarda la verifica della correttezza delle informazioni mi sono basato sui risultati delle esecuzioni dei comandi di OpenSSH.

Il progetto di test è composto dalle stesse classi precedentemente descritte, ma non espone alcun metodo, anzi, dispone di una funzione main che lo rende un normale eseguibile. All'interno di questa funzione main sono contenute le parti di codice necessarie per il testing delle varie classi. I dati utilizzati per l'esecuzione dei test sono stati ricavati da delle precedenti esecuzioni dei comandi di OpenSSH. Ad esempio, per testare la generazione, l'invio e la ricezione di una richiesta di MakeCredential ho utilizzato i parametri forniti da una precedente esecuzione della funzione sk_enroll. L'utilizzo di dati provenienti da precedenti esecuzioni dei comandi di OpenSSH mi ha permesso di testare il funzionamento della libreria con l'autenticatore e di introdurre errori volontari per effettuare il test di risposte con esito negativo.

Tuttavia, questo progetto di test ha permesso solo di verificare il funzionamento dell'invio e della ricezione, oltre che della gestione di errori ricevuti dall'autenticatore. Per verificare invece la correttezza vera e propria delle informazioni ricevute dall'autenticatore, ad esempio i dati ricevuti in seguito alla generazione di una credenziale, sono stati eseguiti direttamente i comandi da terminale di OpenSSH verificandone l'output prodotto. Il progetto di test mi ha inoltre permesso di sfruttare il debugger integrato nell'IDE e semplificare il processo di debugging delle classi.

4.3.1 Comandi OpenSSH

In seguito verrano elencati degli esempi di comandi utili per l'utilizzo della libreria con OpenSSH. Sono inoltre brevemente descritti i parametri

utilizzati. I comandi descritti in seguito sono delle versioni basilari per l'utilizzo di OpenSSH, pertanto i flag indicati sono solo una piccola parte di quelli disponibili. Per questo motivo sono indicati i link ai manuali dei comandi per una visione più approfondita. Si noti che il flag che identifica il path della libreria può essere omesso nel caso in cui sia stata settata la corrispondente variabile d'ambiente come indicato nelle release notes.

Generazione di una credenziale

Con questo comando[9] viene generata sull'autenticatore una nuova credenziale e ne viene salvato un file di OpenSSH contenente l'handle e altre informazioni. Questo file dal punto di vista logico equivale al file contenente una qualsiasi chiave privata. OpenSSH ha infatti un formato proprietario per le chiavi private e lo hanno riutilizzato anche per il salvataggio degli handle [15]. Inoltre, viene salvato un altro file contenente la chiave pubblica. Grazie all'handle è possibile identificare la chiave sull'autenticatore da utilizzare in fase di firma.

```
ssh-keygen -t ecdsa-sk  -f key_filename
        -O user="displayName" -O resident
        -w /path/to/the/library.so
```

- *-t*: tipo di chiave da generare, in questo caso ecdsa-sk (sk indica l'utilizzo di un autenticatore).
- *-f*: nome del file in cui viene salvato l'handle e le informazioni correlate (e.g. user o resident).
- *-O user*: equivale al displayName del CTAP2.
- *-O resident*: indica che la chiave da generare è di tipo resident key.
- *-w*: indica il path alla libreria BLESSH.

[9]https://man7.org/linux/man-pages/man1/ssh-keygen.1.html

Autenticazione

Questo è il comando[10] più conosciuto e utilizzato della suite e serve per autenticarsi presso un server OpenSSH indicando username e chiave da utilizzare.

```
ssh -oSecurityKeyProvider=/path/to/the/library.so
    -i key_filename  username@IP
```

- *-oSecurityKeyProvider*: indica il path alla libreria BLESSH.
- *-i*: nome del file contenente l'handle e le informazioni correlate (e.g. user o resident).
- *username*: username dell'utente del server con cui ci si vuole autenticare.
- *IP*: indirizzo IP del server.

Download resident keys

Per questa operazione sono segnalati due comandi differenti, il primo ha lo stesso comportamento di una normale generazione chiavi ma salva i dati della resident key presente sull'autenticatore all'interno di un file definito da OpenSSH. Il secondo[11] invece esegue il download degli handle e li carica all'interno dell'agent, senza eseguirne il salvataggio su file. Il comportamento attuale di OpenSSH è differente in base ai due comandi. Infatti, attraverso il keygen è possibile ottenere i dati di una sola chiave in quanto in presenza di più chiavi il file viene sovrascritto, mentre attraverso l'add non abbiamo questo problema in quanto le chiavi sono direttamente inserite all'interno dell'agent.

```
ssh-keygen -K -w /path/to/the/library.so
```

```
ssh-add -K -S /path/to/the/library.so
```

- *-K*: indica di eseguire download dei dati delle resident key.
- *-w* e *-S*: indicano per i rispettivi comandi il path alla libreria BLESSH.

[10] https://man7.org/linux/man-pages/man1/ssh.1.html

[11] https://man7.org/linux/man-pages/man1/ssh-add.1.html

Capitolo 5

Conclusioni

L'obiettivo di questo lavoro è stato di sviluppare una libreria di comunicazione che permettesse di espandere le funzionalità offerte dalla suite di OpenSSH, in particolare fornendo i meccanismi base per la comunicazione via Bluetooth Low Energy con un autenticatore FIDO. Nelle prove effettuate durante e alla fine del progetto è stato possibile generare correttamente delle credenziali tramite autenticatore ToothPic e successivamente utilizzarle per autenticarsi in una macchina virtuale attraverso SSH, inoltre è stato (in parte come vedremo) possibile utilizzare delle credenziali già salvate sull'autenticatore. Tuttavia, sono presenti alcune criticità che saranno esposte nella sezione seguente insieme ad eventuali metodologie di risoluzione.

5.1 Limiti e futuri sviluppi

Di base, la libreria offre le funzionalità richieste da OpenSSH, ma analizzando più a fondo il funzionamento sono emersi alcuni problemi dovuti a scelte fatte in fase di progettazione e a implementazioni non dipendenti dal lavoro svolto.

Il primo problema, e sicuramente il più grave è relativo al meccanismo di timeout a livello di comunicazione Bluetooth. In particolare, una volta stabilita la connessione, nel nostro caso in ricezione sfruttando il meccanismo delle notifiche, è presente un timeout di 25 secondi. Questo deriva dal middleware D-Bus [4] utilizzato dalla libreria bluetooth tinyB. Allo scadere del timeout, se la connessione non è stata completata viene

lanciata un'eccezione da D-Bus che pone fine alla comunicazione. Questo può essere riprodotto ad esempio lanciando da OpenSSH il comando per la generazione di una nuova credenziale, e in seguito attendere 25 secondi prima di dare la conferma attraverso lo smartphone. Per via delle scelte implementative del CTAP2 lo smartphone in attesa di conferma da parte dell'utente apre subito una comunicazione inviando dei messaggi di "keep-alive": [1] si tratta di piccoli pacchetti inviati dall'autenticatore a pochi millisecondi uno dall'altro che hanno lo scopo di mantenere aperta la comunicazione segnalando lo stato in cui si trova l'autenticatore (in fase di elaborazione dati o in attesa di azione dell'utente). In teoria, alla ricezione di questi messaggi andrebbe resettato oppure incrementato il timeout, in modo da permettere all'utente di effettuare l'operazione richiesta o all'autenticatore di completare il suo processo di elaborazione, tuttavia occorre comunque definire un tempo limite, come suggerito nel CTAP2. La ricezione di questi pacchetti però, come detto, porta ad occupare la comunicazione attraverso D-Bus il quale dopo il timeout lancerà un'eccezione disconnettendoci dall'autenticatore. Dalla documentazione di D-Bus è presente la possibilità di estendere il timeout sia attraverso un metodo che attraverso un parametro in fase di chiamata dei metodi, tuttavia, questo non è stato implementato dalla libreria tinyB. Se fosse stato possibile modificare il valore di default, avremmo potuto settare un valore molto elevato e gestire un timeout personalizzato tenendo conto del numero di messaggi "keep-alive"ricevuti. Il funzionamento di D-Bus è complicato in quanto sfrutta del codice generato ad hoc per la libreria tinyB e anche effettuando delle modifiche ai sorgenti non è stato possibile aumentare questo timeout. Le soluzioni possibili sono principalmente due: la ricerca di un'altra libreria BLE (dispendioso in quanto andrebbe modificata la classe di comunicazione BLEManager) oppure"riavviare" lo sviluppo di tinyB (come si può vedere dal git, la libreria non viene più sviluppata da ottobre 2017) andando a rigenerare il codice per D-Bus usando una versione aggiornata (questo processo sarebbe ancora più dispendioso in quanto sarebbe richiesta una conoscenza più approfondita di D-Bus e ci si troverebbe a dover mantenere due progetti al posto di uno solo).

Il secondo problema incontrato è nuovamente dovuto alla libreria tinyB.

[1] `https://fidoalliance.org/specs/fido-v2.0-ps-20190130/fido-client-to-authenticator-protocol-v2.0-ps-20190130.html#ble-constants`

È meno grave del precedente ma va comunque preso in considerazione: non è possibile leggere il valore del'MTU negoziata tra client e autenticatore. Su Android è possibile ottenere il valore attraverso una chiamata alle API del Bluetooth, mentre nella libreria tinyB questo non è stato implementato. Il problema si "risolve" sfruttando una specifica definita dal CTAP2, la lettura della caratteristica "fidoControlPointLength"[2] infatti ci permette di ottenere il massimo valore della MTU negoziata in fase di connessione. Questo potrebbe tuttavia portare problemi nel caso il valore letto dalla caratteristica fosse sbagliato nel caso di autenticatori non perfettamente conformi allo standard. La possibilità di lettura dell'MTU direttamente dall'adattatore BLE avrebbe permesso di effettuare una verifica sul valore comunicato dall'autenticatore. Le soluzioni alternative sono analoghe a quelle del problema precedente.

Il terzo e ultimo problema riscontrato è dovuto un'implementazione di OpenSSH che presumibilmente verrà corretta nelle future versioni. All'atto della generazione di una nuova credenziale, se si indica il flag di "user verification" non sarà possibile usare la credenziale generata se si proverà a farne il download dall'autenticatore tramite ssh-add. Analizzando il sorgente dell'agent di OpenSSH si vede che è presente un controllo che indica che il flag non è supportato (quando in realtà questo flag è supportato dalla versione 8.4) ma solo nel caso si utilizzi un middleware esterno. Infatti, nel ramo di utilizzo del middleware interno per autenticatori USB questo controllo non è presente. Come detto, probabilmente verrà sistemato nelle prossime versioni.

Lo sviluppo futuro della libreria, oltre alla correzione dei problemi precedentemente descritti, dovrebbe prevedere l'aggiornamento alle specifiche del CTAP 2.1 quando questo uscirà dalla fase di draft in cui si trova e sarà completamente implementato nell'autenticatore.

[2] https://fidoalliance.org/specs/fido-v2.0-ps-20190130/fido-client-to-authenticator-protocol-v2.0-ps-20190130.html#ble-protocol-overview

Bibliografia

[1] Bluez. http://www.bluez.org/.

[2] Cbor. https://cbor.io/.

[3] cborg. https://github.com/PelionIoT/cborg.

[4] Dbus. https://www.freedesktop.org/wiki/Software/dbus/.

[5] Fido certified products. https://fidoalliance.org/certification/fido-certified-products/.

[6] Fido2. https://fidoalliance.org/fido2/.

[7] Gattlib++. https://github.com/psychogenic/gattlibpp.

[8] Hcitool. https://linux.die.net/man/1/hcitool.

[9] libble++. https://github.com/edrosten/libblepp.

[10] Owasp top ten. https://owasp.org/www-project-top-ten/.

[11] Tinyb. https://github.com/intel-iot-devkit/tinyb.

[12] Toothpic authenticator. https://www.toothpic.eu/authenticator/.

[13] Use of elliptic curve cryptography (ecc) algorithms in cryptographic message syntax (cms). https://tools.ietf.org/html/rfc5753#section-7.2.

[14] Fido technotes: The truth about attestation, 2018. https://fidoalliance.org/fido-technotes-the-truth-about-attestation/.

[15] The openssh private key format, 2018. https://coolaj86.com/articles/the-openssh-private-key-format/.

[16] Client to authenticator protocol (ctap), 2019. https://fidoalliance.org/specs/fido2/fido-client-to-authenticator-protocol-v2.1-rd-20191217.html.

[17] Certicom. Standards for efficient cryptography 1 (sec 1), 2009. https://www.secg.org/sec1-v2.pdf.

[18] McKinsey. Is cybersecurity incompatible with digital convenience?, 2016. https://www.mckinsey.com/business-functions/mckinsey-digital/our-insights/is-cybersecurity-incompatible-with-digital-convenience.

[19] Microsoft. Windows hello for business: What's new in 2017, 2017. https://channel9.msdn.com/events/Ignite/Microsoft-Ignite-Orlando-2017/BRK2076?ocid=cx-blog-mmpc.

[20] NordPass. Most common passwords. https://nordpass.com/most-common-passwords-list/.

[21] Verizon. Data breach investigations report, 2017. https://enterprise.verizon.com/resources/reports/2017_dbir.pdf.

[22] Yubico. Introducing webauthn. https://pages.yubico.com/webauthn-offers-stronger-mfa.

[23] Yubico. State of password and authentication security behaviors report, 2019. https://pages.yubico.com/2019-password-and-authentication-report.html.