Università
Ca'Foscari
Venezia

Corso di Laurea Triennale
in Informatica

Tesi di laurea

WebAuthn API: analisi di un'autenticazione senza password

Relatore
Ch.mo Prof. Stefano Calzavara

Laureando
Francesco Bruno

Matricola
875812

Anno Accademico
2020/2021

Abstract

L'utilizzo di password testuali nei processi di registrazione e autenticazione di un utente presso un servizio web ha accompagnato internet nella sua esistenza. Negli anni '60 Fernando Corbató inventò questa tecnologia, da allora è stata il principale metodo con il quale si effettua l'identificazione di un utente attraverso un elaboratore. Molte tecnologie alternative furono create, un esempio su tutte sono le password grafiche, queste però non sostituirono mai l'utilizzo della password testuale, anche a causa di un'usabilità minore percepita dall'utente utilizzatore.
Lo scopo di questa tesi è quello di presentare il protocollo FIDO2 e nel dettaglio WebAuthn API, una nuova tecnologia alternativa alle password testuali che mira ad eliminare l'utilizzo di esse nelle registrazioni ed autenticazioni effettuate in qualsiasi sito web, con un occhio alla sicurezza ed alla privacy dell'utente usufruente.
Verrà analizzata nel dettaglio l'implementazione di questa nuova soluzione, concentrandosi poi nell'aspetto della sua affidabilità, andando ad approfondire singolarmente le possibili vulnerabilità riscontrate e modellando questa tecnologia all'interno di un software che ha permesso di confermare le criticità trovate in determinate condizioni e la solidità dell'API in altre.

Indice

1 Introduzione **6**
- 1.1 Stato dell'arte . . . 6
- 1.2 Contributi . . . 7
- 1.3 Struttura della tesi . . . 8

2 FIDO2 **9**
- 2.1 Relying Party . . . 11
- 2.2 Autenticatori . . . 11

3 WebAuthn API **14**
- 3.1 Credenziale . . . 14
- 3.2 Attestazione . . . 16
- 3.3 Implementazione . . . 17
  - 3.3.1 Registrazione . . . 18
  - 3.3.2 Autenticazione . . . 26
- 3.4 Problematiche riscontrate . . . 31
  - 3.4.1 Perdita di un autenticatore . . . 31
  - 3.4.2 Compatibilità con i browser . . . 32

4 Sicurezza e privacy di WebAuthn API **34**
- 4.1 Threat model . . . 34
- 4.2 Threat attuabili da un network attacker . . . 34
  - 4.2.1 Man-in-the-middle . . . 34
  - 4.2.2 Replay attack . . . 36
- 4.3 Threat attuabili da un web attacker . . . 38
  - 4.3.1 Phishing . . . 38
  - 4.3.2 Clickjacking . . . 39
  - 4.3.3 Cross-site Scripting . . . 40
  - 4.3.4 Privacy leak . . . 42

5 Analisi formale **46**
- 5.1 Applied Pi Calculus . . . 47
  - 5.1.1 Linguaggio . . . 47
- 5.2 Modellando WebAuthn API . . . 49

5.2.1 Threat Model 53
5.2.2 Processi 53
5.2.3 Cerimonie 54
5.2.4 Formalizzazione 55
5.2.5 Risultati 61

6 Conclusioni **66**

A mio padre, questa è tutta tua.

Capitolo 1

Introduzione

1.1 Stato dell'arte

La registrazione e l'autenticazione di un utente in totale sicurezza è da sempre un punto cardine delicato all'interno dello sviluppo di un'applicazione ed al giorno d'oggi la tecnologia che prevede l'utilizzo di password testuali affiancate ad uno username/email è ancora la più utilizzata dalle persone.
Molte vulnerabilità ed altrettanti attacchi sono emersi nel tempo utilizzando questa tecnologia, uno fra tutti è il *phishing*. Contemporaneamente a ciò, numerose altre tecniche di autenticazione sono state sviluppate in alternativa alla password testuale, un esempio sono le password grafiche ma, nonostante queste nuove tecnologie abbiano iniziato a prendere piede tra gli utenti utilizzatori di dispositivi protetti, non hanno mai veramente spodestato le password testuali, anche per una questione di sicurezza ed usabilità minori percepite dall'utente utilizzatore, ciò implica un ovvio mantenimento di vulnerabilità note che continuano ad essere sfruttate da malintenzionati al fine di violare la privacy e la sicurezza di un individuo.
Per tentare di risolvere questa diatriba ormai fin troppo duratura è stata fondata *Fido Alliance*, organizzazione no profit che ha come mission quella di cambiare la natura dei processi di autenticazione, promuovendo lo sviluppo di standard pubblici che usino soluzioni sicure che riducano o non prevedano password nel loro utilizzo.
Il protocollo **Fast Identity Online 2** (**FIDO2**) è l'ultimo di questi standard e permette un'autenticazione e registrazione completamente *passwordless*[1] o addirittura anche *usernameless*[2] (quest'ultimo solo per certe condizioni che verranno affrontate in questa tesi) a qualsiasi servizio web che lo implementa, garantendo la sicurezza delle informazioni transitanti nel protocollo e mante-

[1] Processo di autenticazione in cui non c'è il bisogno di dover inserire alcuna password da parte dell'utente per poter accedere al suo account in totale sicurezza.

[2] Processo di autenticazione in cui non viene inserito lo username dell'account con cui l'utente vuole accedere.

nendo l'anonimato dell'utente, non richiedendo l'utilizzo di alcun valore che possa identificare la sua identità nella vita reale.
Con queste premesse il non più utilizzo di password in processi di autenticazione sembra gioco fatto, al contrario di quanto detto verranno invece affrontate delle problematiche di sicurezza e non che affliggono questo standard e che potrebbero minarne la sua adozione in favore del mantenimento di sistemi più classici che prevedano l'utilizzo di password testuali come metodo di autenticazione.
La maggior parte delle problematiche di sicurezza o privacy riscontrate sono causate dal non rispetto o dalla mal interpretazione delle linee guida riguardanti l'utilizzo del protocollo da parte di chi ha implementato FIDO2 all'interno di un'applicativo web, solo un piccolo numero invece è causato dall'architettura del protocollo stesso e nonostante ciò, come verrà visto, l'obbligo di utilizzo di crittografia nella comunicazione tra client e server mediante il protocollo HTTPS[3] impedisce e vanifica qualsiasi possibile attacco, garantendo così l'integrità delle informazioni scambiate durante una registrazione e autenticazione.

1.2 Contributi

Con la presentazione dello stato dell'arte, il lettore avrà avuto modo di cogliere attraverso una rapida lettura di come l'utilizzo delle password testuali possa essere evitato tramite l'implementazione di un nuovo protocollo chiamato FIDO2 che, come si può immaginare, ha una grossa responsabilità ed ancora più grosse ambizioni e come annunciato una non corretta implementazione di questo può minare l'integrità dell'intero iter di autenticazione.
La tesi contribuisce quindi, dopo un'approfondita definizione dell'architettura del protocollo, ad analizzare delle possibili vulnerabilità sfruttabili da una persona con intenzioni ostili e dove possibile vengono date delle best practices da seguire per evitare che ciò accada. Oltre a questo, per dare sostegno a quanto scritto viene effettuata un'analisi formale di sicurezza mediante l'utilizzo di un software che ha permesso la modellazione del protocollo sotto forma di uno specifico linguaggio all'interno di un documento, questo viene poi eseguito dal software il quale fornisce dei risultati che, nel caso di FIDO2, hanno confermato quanto trovato precedentemente, verificando inoltre determinate proprietà descritte all'interno del documento ed avvalorando quindi quanto scritto.

[3] Protocollo di comunicazione in una rete internet dove le informazioni scambiate tra mittente e destinatario vengono criptate in modo tale da non renderle leggibili durante il tragitto.

1.3 Struttura della tesi

La tesi presentata, oltre al capitolo introduttivo dove viene fatto conoscere la problematica di un'autenticazione sicura, ne presenta ulteriori 5, all'interno di questi viene affrontata una specifica tematica.
Per iniziare, nel Capitolo 2 viene presentato FIDO2, protocollo che promette un'autenticazione ad applicazioni web totalmente diversa da come l'utente medio è abituato a fare tramite password testuale, dato il non utilizzo di password nell'intero processo di autenticazione.
Successivamente, nel Capitolo 3 viene trattato approfonditamente WebAuthn API, componente di FIDO2, a livello teorico, andando ad analizzare le entità coinvolte e le informazioni che queste si scambiano durante una registrazione o autenticazione, questo necessario per poter capire i capitoli successivi, virando poi al termine di esso in alcune problematiche riscontrate non in ambito di sicurezza che affliggono il protocollo.
Proseguendo, nel Capitolo 4 viene fatta un'analisi della sicurezza dell'API, mantenendosi sempre a livello teorico, basandosi su quanto dichiarato nella documentazione della libreria e da quanto riscontrato dall' implementazione del protocollo in ambienti di sviluppo.
Avendo trattato le vulnerabilità che potrebbero minare l'integrità del protocollo e la privacy di un utente, nel Capitolo 5 si è confermato quanto detto in quello precedente riguardo una particolare tipologia di attacco, cioè quella che prevede un *network attacker*, questa volta attraverso l'utilizzo di uno strumento software, *ProVerif*, il quale ha permesso di modellare WebAuthn API in un preciso linguaggio all'interno di un file che, una volta eseguito simulando un attaccante, ha permesso di dimostrare importanti proprietà introdotte all'interno del capitolo che verificano l'integrità dei dati e non solo.
Concludendo, l'ultimo capitolo chiude la tesi facendo un riassunto di quanto riscontrato dall'analisi di sicurezza sia teorica che via software, facendo poi delle riflessioni su quanto ottenuto.

Capitolo 2

FIDO2

Come introdotto nel primo capitolo, FIDO2 è un protocollo con un grande compito da compiere e che lascia grandi aspettative in ambito di sicurezza informatica, questo perché promette autenticazioni e registrazioni robuste, sicure e che tutelino l'utente dal punto di vista della privacy grazie al non utilizzo di qualsiasi tipologia di password, incrementando anche l'usabilità percepita dall'utilizzatore.
Per permettere ciò sono previsti 3 attori:

- Un *client*, cioè l'utente con il proprio browser;
- Un *server* con il relativo sito web su cui l'utente vuole registrarsi o autenticarsi, vedi paragrafo 2.1;
- Un *autenticatore*, vedi paragrafo 2.2;

Questi 3 cooperano tra di loro al fine di creare una *credenziale* da associare all'utente e permettere quindi ad esso di potersi autenticare in momenti successivi tramite l'utilizzo di questa. Ciò è possibile grazie alla presenza dell'autenticatore, il quale attraverso la verifica della persona tramite riconoscimento biometrico o PIN[1], crea una credenziale mediante l'uso di crittografia asimmetrica[2], associandola univocamente all'utente ed al dominio del sito web su cui questo sta operando, firmandola in maniera tale che sia facilmente convalidabile da parte del server (vedi figura 2.1).
In dettaglio, FIDO2 è suddiviso in due sotto-protocolli:

- ***WebAuthn***, una browser API Javascript (JS) estensione di Credential Management API, entrambe sviluppate da *World Wide Web Consor-*

[1] Mentre una password testuale è collegata ad un singolo account, un PIN è riconducibile all'autenticatore, quindi tutte le credenziali create da un autenticatore A avranno lo stesso PIN

[2] Tipo di crittografia in cui sono previste due chiavi, rispettivamente pubblica e privata, le quali permettono di cifrare o decifrare una qualsiasi informazioni, attestandone anche la paternità.

tium (W3C)[3] che funge da interfaccia tra applicazioni web ed autenticatori, permettendo a queste due entità di comunicare tra loro cosicché un utente possa usare i dispositivi posseduti per accedere o registrarsi a servizi senza l'ausilio di password;

- ***Client to Authenthicator Protocol 2*** (*CTAP2*), protocollo a basso livello sviluppato da FIDO Alliance che permette la comunicazione tra autenticatori roaming[4] ed il dispositivo con cui si sta effettuando un'autenticazione o registrazione, consentendo quindi l'interazione con browser che supportano WebAuthn API. Esso è composto essenzialmente in tre parti:

  1. *Authenticator API*, ogni operazione che può fare l'autenticatore viene astratta e rappresentata attraverso un metodo contenuto all'interno di questa libreria;
  2. Definizione della codifica dei messaggi, descrive il processo di codifica nel formato *Concise Binary Object Representation* (CBOR)[5] che viene applicato su una richiesta da passare all'autenticatore nel momento in cui viene invocato un metodo all'interno di Authenticator API;
  3. Vincoli riguardo il mezzo di trasporto, per ogni mezzo di trasporto (es. USB, NFC, Bluetooth) sono presenti dei vincoli da rispettare per il messaggio da trasmettere;

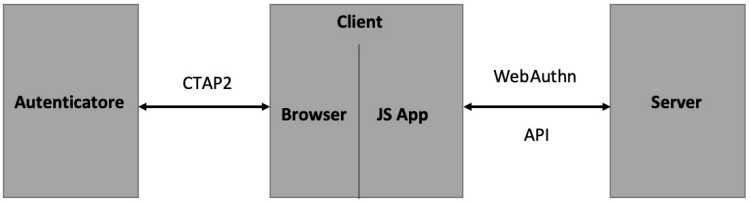

Figura 2.1: *Illustrazione di autenticazione tramite FIDO2.*

Vengono ora definiti dettagliatamente due dei tre attori presenti all'interno dei processi di autenticazione e registrazione che vengono svolti attraverso FIDO2.

[3] Organizzazione non governativa mondiale.
[4] Autenticatore esterno non vincolato ad un singolo dispositivo.
[5] Formato di serializzazione di informazioni il cui risultato è un insieme di dati di dimensione estremamente piccolo e non leggibile da una persona, vedi [9].

2.1 Relying Party

Il termine *Relying Party* (RP) identifica l'entità posseditrice del sito web che usa WebAuthn API e viene spesso intercambiabilmente utilizzato per identificare il sito web di sua proprietà.
L'implementazione del RP consiste in script lato client che invocano metodi definiti all'interno della libreria, ed ulteriori programmi lato server, i quali hanno il compito di verificare e memorizzare i dati necessari per il funzionamento della piattaforma web su cui l'API è eseguita.

2.2 Autenticatori

Rispetto ad una classica autenticazione con password in servizi web, dove tipicamente sono presenti solo l'utente con il proprio dispositivo, il client, ed un sito web ospitato in un server, in un'autenticazione tramite FIDO2 entra in gioco un nuovo soggetto, l'**autenticatore**, entità facente parte della categoria "*Something Possessed*[6]".
L'autenticatore è un dispositivo che generalmente ha il compito di verificare l'identità di una persona nel caso in cui questa voglia accedere o registrarsi ad un servizio soggetto ad autenticazione. Nel caso in cui la verifica abbia esito positivo, l'autenticatore genera ed associa all'utente una credenziale, firmandola, questo solo se richiesto dal Relying Party (nel caso in cui sia in un processo di registrazione), o permette l'utilizzo di credenziali già esistenti (nel caso in cui l'utente sia in un processo di autenticazione).
Esistono due tipi di autenticatore:

- **Cross-Platform**: chiavette fisiche esterne al dispositivo su cui l'utente sta effettuando la richiesta al servizio (vedi Figura 2.2);

Figura 2.2: *Esempio di autenticatore Cross-Platform;*

[6] Un utente può provare la propria identità attraverso un dispositivo che possiede.

- **Platform**: dispositivi che risiedono all'interno del calcolatore che l'utente sta utilizzando (es. Touch-ID o Windows Hello);

È bene sottolineare che l'autenticazione può non coincidere con la verifica dell'utente.
In particolare:

- In un'autenticazione classica con password, il modo in cui l'utente si identifica al device è lo stesso con cui il device prova la sua identità al server su cui risiede il servizio richiesto dall'utente, in questo caso quindi *AUTENTICAZIONE == VERIFICA DELL'UTENTE* (vedi Figura 2.3).

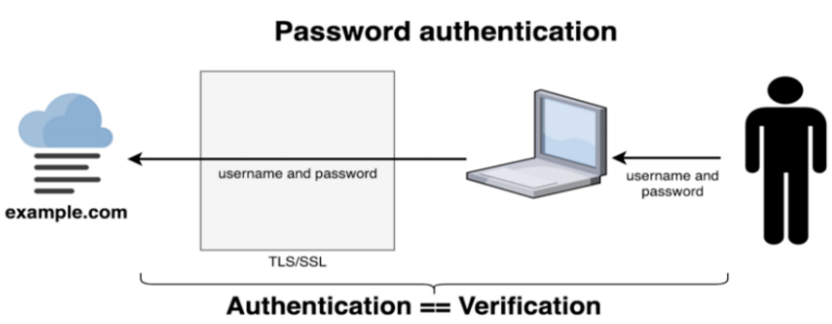

Figura 2.3: *Illustrazione di un'autenticazione con l'ausilio di password.[1]*

- In un'autenticazione passwordless, l'utente verifica la propria identità attraverso l'autenticatore (es. riconoscimento biometrico) e quest'ultimo genera un'asserzione contenente una serie di informazioni che permettono al server di verificare la validità della risposta creata dall'autenticatore. In questo caso quindi *AUTENTICAZIONE ≠ VERIFICA DELL'UTENTE* (vedi Figura 2.4).

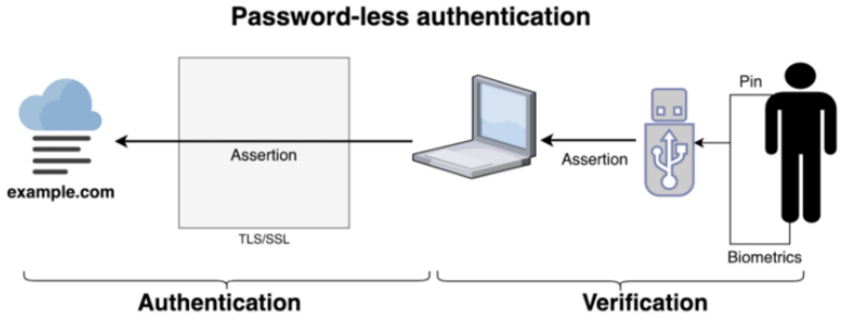

Figura 2.4: *Illustrazione di un'autenticazione senza l'ausilio di password.[1]*

Indipendentemente dal tipo di autenticatore, inoltre, questi si possono distinguere in base alla capacità di memorizzazione che offrono, cioè, un dispositivo può o meno avere la possibilità di salvare al suo interno tutte le informazioni associate ad una credenziale.
In particolare, se un autenticatore dovesse avere la capacità di salvare al suo interno l'intero set di dati, alleggerirebbe il carico di memoria che il server altrimenti dovrebbe gestire, essendo che le informazioni andrebbero salvate buona parte dentro questo, avendo inoltre la possibilità di avere non solo un'autenticazione passwordless ma addirittura usernameless; come grosso svantaggio, invece, il numero di credenziali totali salvabili all'interno dell'autenticatore calerebbe drasticamente (varia a seconda del dispositivo ma mediamente non superano le 40 unità). In contrasto, un autenticatore che salva le informazioni all'interno del server permette un numero illimitato di credenziali memorizzabili al suo interno ma perde la possibilità di avere un'autenticazione usernameless.
Si sottolinea che se un autenticatore ha la capacità di salvare tutti i dati di una credenziale, può comunque supportare una registrazione ove i dati vengono memorizzati su server, questo infatti è a discrezione dell'entità gestrice del sito web.

Capitolo 3

WebAuthn API

Come introdotto nel Capitolo 2, WebAuthn è una browser API Javascript che compone il protocollo FIDO2 e che permette di interfacciare autenticatori e applicazioni web, così da permettere all'utente di effettuare registrazioni ed autenticazioni passwordless o addirittura usernameless. Oltre a ciò, si può implementare tale tecnologia anche come secondo fattore di sicurezza in una Two Factor Authentication (2FA),[1] a scapito di soluzioni meno sicure come per esempio gli SMS[2], questo è possibile grazie alla compatibilità della libreria con Universal Second Factor (U2F)[3].
Prima di studiare come avviene una registrazione o un'autenticazione utilizzando questa API, viene data una definizione a due importanti componenti facenti parte del protocollo e che verranno spesso incontrati nella lettura di questa documento.

3.1 Credenziale

Una credenziale in WebAuthn API è l'insieme di informazioni definite dall'autenticatore durante una registrazione, salvate ed usate poi da esso in un'autenticazione di un utente in modo da permettere al Relying Party di validare la sua identità, questo è permesso tramite l'utilizzo di crittografia asimmetrica, quindi con la presenza di una chiave pubblica e privata.
Un'*asserzione* è l'insieme dei dati che il client fornisce al server per dimostrare l'autenticità dell'identità che l'utente dice di avere, essa viene creata con l'aiuto dell'autenticatore usufruendo di una particolare struttura che viene associata univocamente al termine della registrazione alla persona registrata

[1] Processo di autenticazione in cui è richiesto all'utente di presentare diverse prove per potersi autenticare, ad esempio una password ed un codice inviato sulla email del fruitore del servizio.

[2] Gli SMS sono vulnerabili a *Sim Swapping*. [3]

[3] Protocollo creato da Google e Yubico che punta ad utilizzare dati biometrici o chiavi di sicurezza come secondo fattore di sicurezza, a discapito di altre soluzioni come codici inviati via email o sms.

ed al sito web su cui il processo sta avendo atto, questa struttura è chiamata **Public Key Credential Source** e contiene:

- Tipologia di chiave salvata, di default ha valore "*public-key*";
- ID della credenziale cioè l'identificativo che riconosce univocamente la credenziale tra tutte quelle memorizzate;
- Chiave privata della credenziale, cioè la chiave associata ad essa con cui firmare un'asserzione al momento di un'autenticazione;
- *Relying Party ID* cioè una stringa univocamente identificativa per il Relying Party;
- Identificativo dell'utente registrato associato alla credenziale;
- Altre informazioni usate dall'autenticatore per popolare sua interfaccia grafica (se la possiede), un esempio è "*displayName*", attributo che identifica un nome utilizzabile a video per identificare un utente.

È bene sottolineare che, nonostante il termine di credenziale sia associabile all'asserzione che l'autenticatore genera e che poi fornisce al client, lo stesso termine viene intercambiabilmente usato all'interno delle linee guida dell'API ed all'interno di questo documento anche per identificare la *Public Key Credential Source* associata ad un utente ed il relativo sito web su cui si è registrato, quindi il suo significato cambia a seconda del contesto in cui questo termine è utilizzato.
Ogni credenziale è collegata all'autenticatore che l'ha generata, quindi solo questo potrà creare delle asserzioni per essa.
Non fa parte di *Public Key Credential Source* la chiave pubblica associata alla relativa credenziale ma ovviamente tiene con essa uno stretto legame poiché tramite la chiave pubblica, salvata al termine della registrazione all'interno del Relying Party, quest'ultimo riesce a validare qualsiasi credenziale firmata tramite la chiave privata strettamente salvata all'interno dell'autenticatore e facente parte della struttura da cui la credenziale attinge i propri dati.

Tipi di credenziale

Si possono distinguere due tipi di credenziali in base a come viene memorizzato *Public Key Credential Source*, questo influenza il modo in cui viene svolto il processo di autenticazione a seconda del tipo scelto.

- Se l'autenticatore ha capacità di salvare *Public Key Credential Source* al suo interno e questa viene effettivamente memorizzata all'interno della sua memoria, allora l'elemento salvato prende il nome di **Client-side Credential**, **Client-side discoverable Credential** o **Client-side discoverable Public Key Credential Source**, solo in questo

caso l' autenticatore ha salvato all'interno della propria memoria una mappa[4] che collega *(rpId*[5]*, [userHandle*[6]*]) ↦ Public Key Credential Source*, così da permettere il recupero della struttura quando l'utente richiede di effettuare un'autenticazione.

- Se *Public Key Credential Source* viene memorizzata nel server e non all'interno della memoria dell'autenticatore, l'elemento salvato prende il nome di **Server-side Public key Credential Source** o **Server-side Credential** e solo in questo caso l'ID della credenziale è il valore cifrato di *Public Key Credential Source*, effettuato tramite crittografia simmetrica ed è reversibile solo dall'autenticatore creatore della struttura, cosicché esso possa ricavare l'oggetto originale con i relativi campi durante un'autenticazione nel caso in cui riesca a decifrarlo se fornito dal client.

Nel caso di Server-side Credentials, quindi, il client durante un processo di autenticazione deve fornire all'autenticatore, oltre all'ID del Relying Party su cui si vuole effettuare l'operazione, anche l'identificativo della credenziale in modo che esso possa ottenere la chiave privata utile all'autenticatore per creare l'asserzione necessaria per garantire l'accesso all'utente.
Viceversa, utilizzando Client-side Credentials, avendo l'intero set di informazioni salvate all'interno dell'autenticatore, quando il client decide di effettuare un processo di autenticazione non dovrà fornire all'autenticatore alcun identificativo relativo ad una credenziale, ma solo quello identificante il Relying party, questo perchè l'autenticatore permette all'utente di scegliere, attraverso un pop-up[7], con quale credenziale accedere nel caso ce ne siano multiple associate all'ID del Relying Party, quindi, non solo l'autenticazione è passwordless, ma diventa anche usernameless.

3.2 Attestazione

Il Relying Party può avere diverse policy riguardanti l'attendibilità di un autenticatore, interessandosi o meno al fatto se concedere la creazione di credenziali da dispositivi più o meno affidabili[8].
WebAuthn API permette di gestire questa scelta, implementando la possibilità durante un processo di registrazione di far recapitare al Relying Party una dichiarazione di attestazione firmata la quale dimostra, se verificata, le

[4] Struttura dati che memorizza informazioni accessibili attraverso una chiave, le informazioni saranno memorizzate al suo interno quindi come: chiave ↦ valore

[5] Identificativo univoco associato all'entità che gestisce il sito web su cui sono state create le credenziali.

[6] Identificativo dell'utente associato alla credenziale.

[7] Finestra a scomparsa che compare a video nel computer in un certo istante.

[8] Un autenticatore è considerato affidabile nel momento in cui possiede tutte le proprietà che dice di avere.

proprietà e le qualità di un autenticatore, ma anche la paternità delle credenziali create.
Ci sono diversi *tipi* e *formati* che caratterizzano un'attestazione (**N.B.**: tipi $\neq$ formati) e mentre il primo definisce la semantica, fornendo un proprio modello di attendibilità, il formato, invece, definisce una precisa sintassi. Ogni tipo ha le proprie compatibilità con i diversi formati ed a seconda della sintassi usata, il processo di verifica che dovrà effettuare il Relying Party per accertarsi dell'affidabilità del creatore della credenziale fornita cambierà.
L'autenticatore al momento della creazione di una credenziale sceglierà il tipo dell'attestazione in base a quanto suggerito dal Relying Party, vedi il parametro "*attestation*" presente in *PublicKeyCredentialCreationOptions*, entrambi spiegati nel paragrafo 3.3.1.
Un esempio di tipo è "*Basic attestation*" dove l'autenticatore firmerà le credenziali attraverso una chiave privata di attestazione, questa solitamente associata, ad un batch di dispositivi (solitamente 100'000)[9] e ad ogni firma fatta con la chiave privata, la corrispettiva pubblica per poterla verificare è inserita all'interno di un certificato presente dentro la dichiarazione.
Altri esempi possono essere "*Self attestation*", dove l'autenticatore non dispone di una propria chiave privata per poter firmare l'attestazione e firma la stessa attraverso l'utilizzo della chiave privata della credenziale generata su cui sta attestando la proprietà (più insicuro rispetto a "Basic attestation" poichè prova solamente la paternità della credenziale che si sta creando e non l'affidabilità dell'autenticatore con cui lo si sta facendo) o "*No attestation stament*", usata laddove il Relying Party non è interessato alla verifica dell'attendibilità di un autenticatore utilizzato.

3.3 Implementazione

WebAuthn API è nativamente implementata all'interno dei browser compatibili, vedi paragrafo 3.4.2, ciò permette al Relying Party di usufruire delle funzionalità di questa libreria per processi di registrazione ed autenticazione attraverso delle semplici chiamate a due metodi definiti in Credential Management API:

- **navigator.credentials.create()** - Il client invoca questo metodo asincrono quando desidera creare delle nuove credenziali da associare all'utente;

[9] Questo perché se la chiave privata dovesse essere compromessa, un batch troppo piccolo potrebbe compromettere l'identità di una persona perché associabile ad essa, un batch troppo grande invece sarebbe oneroso per l'azienda produttrice in termini economici da sostituire, vedi [7].

- **navigator.credentials.get()** - Il client invoca questo metodo asincrono quando desidera utilizzare delle credenziali già esistenti con cui autenticare l'utente;

Questi processi in cui vengono chiamati i metodi definiti qui sopra prendono il nome di **cerimonie** e vengono effettuati, per esplicita richiesta dalle linee guida, in contesti sicuri, quindi laddove client e server risiedano all'interno dello stesso dispositivo (es. localhost) o dove dovesse esserci comunicazione criptata in una rete utilizzando il protocollo HTTPS.

3.3.1 Registrazione

La cerimonia di registrazione è l'importante processo che permette ad un utente la creazione di un account all'interno di un sito web, questo effettuato senza l'ausilio di password ed al termine della quale sia l'autenticatore che il Relying Party avranno tutti le informazioni per la creazione o validazione di future credenziali durante delle autenticazioni di una persona.
L'intero processo prevede diverse interazioni tra client $\leftrightarrow$ server e client $\leftrightarrow$ autenticatore attraverso le quali avviene uno scambio di informazioni utili al compimento della cerimonia stessa (vedi Figura 3.1).

Viene ora spiegato dettagliatamente come avviene una registrazione, analizzando così tutte le informazioni scambiate tra gli attori coinvolti e che sono mostrate in Figura 3.1.
La cerimonia inizia nel momento in cui l'utente, dopo aver compilato il relativo form per registrarsi al servizio web, invia al server una richiesta allegando quanto compilato, quindi facendo conoscere a questo le proprie intenzioni di voler creare un account/una credenziale, attendendo poi una risposta da esso.
Il RP, avendo ricevuto le intenzioni dell'utente, crea un oggetto ***PublicKeyCredentialCreationOptions***, estensione del dizionario *CredentialCreationOptions* definito in Credential Management API. Questo oggetto viene appositamente creato lato server per motivi di sicurezza, essendo che contiene parametri utili per l'autenticatore e per l'intera cerimonia che se conosciuti minerebbero l'affidabilità della stessa. In dettaglio, l'oggetto *PublicKeyCredentialCreationOptions* ha al proprio interno i seguenti parametri:

- **rp**, oggetto obbligatorio contenente l'ID del Relying Party, in particolare questo è una stringa che lo identifica univocamente. Di default il suo valore è il dominio della pagina su cui sta effettuando la registrazione l'utente, può assumere valore pari ad un suo suffisso, non ci sono restrizioni sulla porta utilizzata ed il protocollo usato deve essere HTTPS (se diverso da localhost);

- **user**, oggetto obbligatorio contenente un array di byte non vuoto rappresentante un ID univoco associato all'utente che sta effettuando la

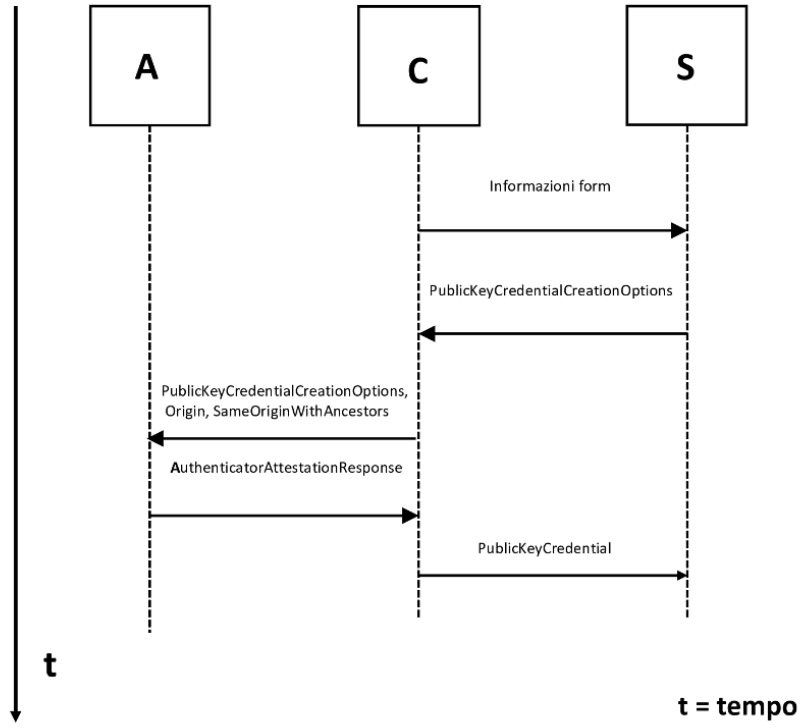

Figura 3.1: *Illustrazione di una cerimonia di registrazione. A=autenticatore, C=client, S=server*

cerimonia di registrazione, non è pensato per essere mostrato all'utente a cui è associato. È caldamente consigliabile che questo identificativo sia pseudocasuale così da anonimizzare la persona, in linea con la filosofia del protocollo, perciò non deve contenere informazioni associabili ad un utente e deve essere lungo massimo 64 byte. Oltre a questo ID, viene salvato anche *DisplayName*, stringa pensata per essere mostrata a video e che contiene un nome per l'account che si creerà se la cerimonia termina correttamente; questa stringa potrebbe contenere metadati come la lingua e la direzione di lettura della stessa, se non presenti, il Relying Party dovrebbe fornire queste informazioni per una corretta visione di questo parametro;

- **challenge**, oggetto di tipo BufferSource[10] contenente un array di byte generato randomicamente ad ogni cerimonia per evitare replay attacks

[10] Tipo usato per rappresentare oggetti che sono ArrayBuffer, al cui interno si trova un array di byte di dimensione fissa, oppure qualsiasi tipo di oggetto avente tipo TypedArray in Javascript.

(vedi paragrafo 4.2.2), deve essere lungo almeno 16 byte e contenere abbastanza entropia da rendere impossibile indovinare il proprio valore.

- **pubKeyCredParams**, parametro obbligatorio il cui valore è un array dove i suoi elementi racchiudono informazioni sulle proprietà che il RP richiede che le credenziali create dall'autenticatore abbiano, come il tipo (stringa che alla scrittura di questo documento assume solo valore "*public-key*") e l'algoritmo utilizzato, il quale deve corrispondere ad uno degli algoritmi presenti in COSE[11]. Gli elementi di questo array sono ordinati in base alla preferenza dell'algoritmo da utilizzare,

- **timeout**, parametro facoltativo che identifica tramite numero i millisecondi il tempo massimo che il chiamante aspetta per il completamento dell'operazione, il browser potrebbe sovrascrivere questo valore.
  Si consiglia:

  - Se è richiesta o preferita la verifica dell'utente, allora il timeout dovrebbe variare tra 30000 e 600000, raccomandato 300000;
  - Se non è richiesta la verifica dell'utente, allora il timeout dovrebbe variare tra 30000 e 180000, raccomandato 120000;

- **excludeCredentials**, parametro facoltativo rappresentante una lista di credenziali identificate dal loro ID ed associate all'ID dell'utente, di default è posto vuoto. Questo parametro è inteso per limitare la creazione di credenziali da un utente per uno specifico autenticatore;

- **authenticatorSelection**, parametro facoltativo contenente dei vincoli per l'autenticatore, in particolare potrebbe contenere uno o più di questi attributi:

  - **authenticatorAttachment**, stringa facoltativa che permette di specificare se il tipo di autenticatore deve essere ristretto a "*cross-platform*" o "*platform*", se assente entrambi sono ammessi;
  - **residentKey**, stringa facoltativa che può assumere valore "*discouraged*", "*preferred*" o "*required*" a seconda della preferenza del RP riguardo la creazione di *Client-side Credential*;
  - **requireResidentKey**, valore booleano facoltativo posto di default con valore "*false*", deve essere posto "*true*" se e solo se l'attributo residentKey precedentemente presentato abbia valore "*required*";
  - **userVerification**, stringa facoltativa posta di default a "*preferred*", indica il requisito del RP di richiedere o meno la verifica

[11] Specifica che descrive come gestire la crittografia usando il formato CBOR come formato di serializzazione, vedi [4].

dell'utente da parte dell'autenticatore. Se non richiesta, l'autenticatore richiederà solo un test di presenza, può assumere valori "*required*", "*preferred*" o "*discouraged*".

- **attestation**, stringa opzionale che indica la preferenza del RP riguardo la trasmissione o meno di un'attestazione da parte dell'autenticatore, in particolare:
  - "*none*", valore di default, il RP non è interessato all'attestazione dell'autenticatore;
  - "*direct*", il RP è interessato a ricevere un'attestazione dall'autenticare che sia verificabile e vuole che questa sia così come l'autenticatore l'ha creata;
  - "*indirect*", il RP è interessato a ricevere un'attestazione dall'autenticare che sia verificabile, ma permette al browser di decidere come ottenerla o di rimpiazzarla con un'attestazione di tipo più "privacy-friendly";
  - "*enterprise*", il RP è interessato a ricevere un'attestazione dall'autenticatore che sia verificabile, essa può contenere informazioni che identificano univocamente l'autenticatore stesso;
- **extensions**, parametro opzionale contenente ulteriori elaborazioni richieste per il client o per l'autenticatore, sono definite in "*WebAuthn Extension Identifiers*" situato in [13].

```
publicKeyCredentialCreationOptions = {
  //RANDOM BINARY CHALLENGE
  challenge: "eC8H5nmolg-bpr3fkVq-0JrKNMDDuwE3q1i1k1KE8",
  rp: {id: "https://esempio.it"},
  user: {
    id: "bpr3fkVq-0JrKNMDDu",
    displayName: "Francesco"
  },
  authenticatorSelection: {
    userVerification: "preferred",
    authenticatorAttachment: "cross-platform"
  },
  attestation: "direct",
  timeout: 300000,
  excludeCredentials:[{
    type:"public-key",
    id:"3f0_EHd3rkWe4_LALY_LIcLy2mInAxi42A7wA7GUgmKZLVs",
    transports:["usb","nfc","ble","internal"]
  }],
  pubKeyCredParams: [ { type: "public-key", alg: -7} ]
}
```

Codice 3.1: *Esempio di PublicKeyCredentialCreationOptions*

Una volta creato *PublicKeyCredentialCreationOptions* (vedi Codice 3.1):

1. Il server ritorna questo oggetto al client come risposta alla sua richiesta;
2. Il client invoca il metodo *navigator.credentials.create()*, passando come parametri l'oggetto ottenuto nel punto 1), *origin*, tupla identificante l'indirizzo del sito web su cui l'utente sta effettuando la registrazione e *sameOriginWithAncestors*, parametro booleano il quale assume valore *true* se e solo se *origin* coincide con l'indirizzo del sito web che permette la visione del form di registrazione (si ipotizza la presenza di un iframe[12] all'interno di un sito web A che permette la visione di una pagina appartenente ad un altro sito web, B). Per questioni di sicurezza ha sempre valore *true* poiché non è permessa la creazione di credenziali attraverso cross-origin iframe, vedi paragrafo 4.3.2;
3. Il browser controlla ed elabora quanto ricevuto;
4. Il client cerca un autenticatore collegato e si connette con questo;
5. All'interno del metodo richiamato nel punto 2), il browser ne invoca un altro facente parte del protocollo CTAP2, *authenticatorMakeCredential*, il quale permette di far ricevere all'autenticatore un'elaborazione dei dati contenuti nell'oggetto ricevuto dal server nel punto 1);
6. L'autenticatore esegue una verifica dell'utente o un test di presenza a seconda di quanto suggerito dal Relying Party nell'oggetto *publicKeyCredentialRequestOptions* (vedi Codice 3.1 e Figura 3.2);

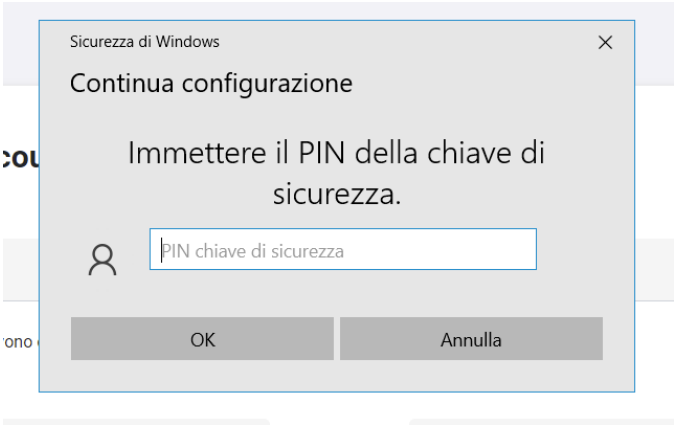

Figura 3.2: *Esempio di verifica di un utente tramite autenticatore Cross-Platform durante una registrazione.*

[12] Elemento HTML che permette la visione di una pagina web all'interno di un riquadro situato in una pagina web differente.

7. Se tutto è andato a buon fine, l'autenticatore genera le chiavi da associare all'utente e *attestationObject* (vedi Figura 3.3), oggetto readonly crittograficamente protetto dalla manomissione e di tipo *ArrayBuffer* codificato in CBOR. Al suo interno è contenuta la chiave pubblica, una dichiarazione di attestazione verificabile dal RP se richiesta da esso ed altri metadati dell'autenticatore utili alla verifica della registrazione. Questo oggetto viene inserito in un ulteriore oggetto contenente altri valori, come per esempio *clientDataJSON*, array di byte contenente un'elaborazione dei dati personali dell'utente che si sta registrando, questo è stato creato nel punto 3) di codesto elenco ed è stato passato all'autenticatore insieme al resto dei dati ricevuti. Tutto questo viene tornato al client;

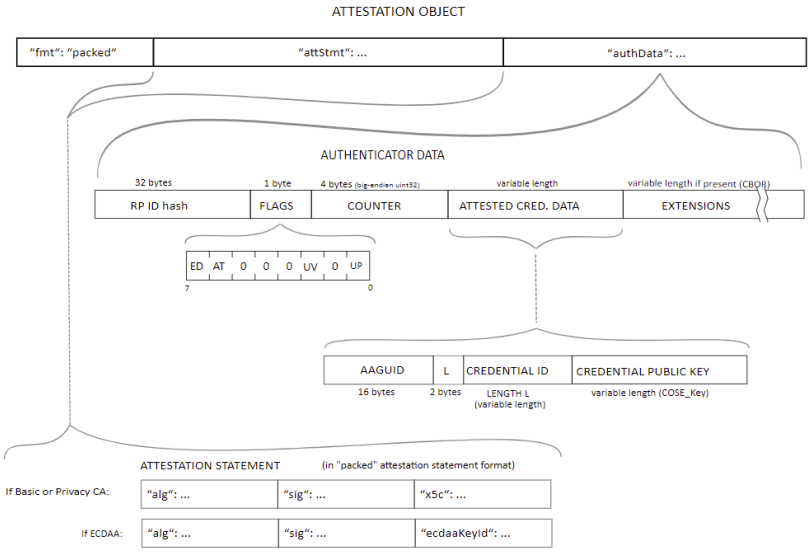

Figura 3.3: *Composizione di attestationObject.[12]*

8. Una volta che il browser riceve la struttura, esso la elabora e serializza, ottenendo un oggetto, che sarà l'effettiva credenziale, rispettante l'interfaccia *PublicKeyCredential* (vedi Codice 3.2), fornendolo poi al server per validazione.

```
PublicKeyCredential {
  id: "3f0_EHd3rkWe4_LALY_LIcLy2mInAxiA7wA7GUgmKZLVs",
  rawId: ArrayBuffer(59),
  response: AuthenticatorAttestationResponse {
    clientDataJSON: ArrayBuffer(112),
    attestationObject: ArrayBuffer(299),
  },
  type: "public-key"
}
```

Codice 3.2: *Esempio di oggetto che rispetta l'interfaccia PublicKeyCredential*

Analizzando la credenziale ricevuta dal client si possono scorgere diversi parametri contenuti al suo interno utili alla validazione ed alla memorizzazione della stessa, elencandoli:

- **id**, stringa contenente l'identificatore delle credenziali appena create dall'autenticatore;
- **rawId**, identificatore delle credenziali scritto in forma binaria;
- **response**, oggetto rispettante l'interfaccia *AuthenticatorAttestationResponse* e contenente:
  - **clientDataJSON**, il cui valore è un oggetto di tipo ArrayBuffer e serializzato in JSON in cui sono contenute alcune informazioni che il browser ha passato all'autenticatore. In particolare, al suo interno si trovano la challenge inviata dal RP ed *origin*, un parametro dal nome *crossOrigin* avente come valore l'opposto di quanto contenuto in *sameOriginWithAncestors* ed un oggetto facoltativo descrivente lo stato del protocollo Token Binding[13], inoltre è presente un parametro *type* avente valore sempre pari a "*webauthn.create*", vedi Codice 3.3;
  - **attestationObject**, il cui valore è un oggetto di tipo ArrayBuffer protetto crittograficamente, contenente dati utili alla verifica della registrazione, come metadati relativi all'autenticatore e un'attestazione, se richiesta dal RP, vedi Codice 3.4 e Figura 3.3;
- **type**, stringa il cui valore è sempre "*public-key*" e rappresenta il tipo di credenziale rappresentata dal corrente oggetto in cui è contenuto l'attributo;
- **extensions**, mappa facoltativa i cui elementi al suo interno hanno forma *identificativo delle estensioni* $\mapsto$ *output generato*;

[13] Protocollo utile ad aumentare la sicurezza di un sito web che fa uso di token di sicurezza in un'autenticazione.

```
{
  challenge: "eC8H5nmolg-bpr3fkVq-0JrKNMDDuwE3q1i",
  origin: "https://esempio.it",
  type: "webauthn.create",
  crossOrigin:false

}
```

Codice 3.3: *Esempio di clientDataJSON.*

```
{
  authData: Uint8Array(196),
  fmt: "fido-u2f",
  attStmt: {
    sig: Uint8Array(70),
    x5c: Array(1),
  },
}
```

Codice 3.4: Esempio di *attestationObject*

In dettaglio, concentrandosi su *attestationObject*, al suo interno sono situati:

- **authData**, il cui valore è un array di byte in cui sono codificati dati relativi all'autenticatore (es. $AAGUID$[14]), al relying party, la chiave pubblica, l'ID delle credenziali create, flag che specificano se fosse richiesta la verifica dell'utente o solamente un test di presenza, un contatore chiamato *signCount* (vedi paragrafo 4.2.2) posto inizialmente a 0 che indicherà quante cerimonie di autenticazione sono state effettuate usando quella credenziale attraverso quell'autenticatore (questa funzionalità non è sempre supportata da tutti gli autenticatori) ed altri metadati utili alla verifica da parte del server.
- **fmt**, il cui valore rappresenta la sintassi che avrà l'attributo successivamente spiegato *attStmt*, vedi paragrafo 3.2;
- **attStmt**, il cui valore dipende da quanto definito nel valore di *fmt*, generalmente contiene una firma fatta attraverso la chiave privata di attestazione dell'autenticatore (tranne nel caso di "*self-*

[14] Identificativo descrivente marca e modello dell'autenticatore.

attestation") ed una catena di certificati, ciuascuno codificato in formato X.509[15].

Il server ha una serie di operazione da fare sull'oggetto ricevuto in modo da verificare la sua integrità e quindi di accertarsi che l'entità che sta effettuando la registrazione sia chi dice di essere.
Una volta eseguiti questi controlli, il server può finalmente registrare l'utente, associando ad esso la chiave pubblica della credenziale e memorizzando questa al proprio interno. É importante che questi controlli vengono fatti per non avere una cerimonia di registrazione inconsistente.

3.3.2 Autenticazione

La cerimonia di autenticazione è il processo con cui un utente riesce ad entrare in un'area protetta di un sito web o commettere particolari azioni solo e solamente dopo essersi identificato al server.
Questo è permesso grazie alla creazione da parte dell'autenticatore di un'asserzione contenente una credenziale con cui l'utente vuole accedere, questa viene poi convalidata dal server grazie alle informazioni che ha ricevuto nella precedente cerimonia di registrazione.
Come per la registrazione, l'intero processo di autenticazione prevede diverse interazioni tra client $\leftrightarrow$ server e client $\leftrightarrow$ autenticatore attraverso le quali avviene uno scambio di informazioni utili al compimento della cerimonia stessa (vedi Figura 3.4). Viene ora analizzato come avviene un'autenticazione, andando ad approfondire il contenuto delle informazioni scambiate tra gli attori coinvolti, entrando quindi nel dettaglio di quanto mostrato in Figura 3.4.
La cerimonia inizia nel momento in cui un utente compila il relativo form o preme un pulsante facendo capire le intenzioni al Relying Party di volersi autenticare. Similmente a quanto accade durante una registrazione, il RP, ricevendo le intenzioni dell'utente crea un oggetto **publicKeyCredentialRequestOptions**, questo contiene:

- **challenge**, oggetto di tipo BufferSoure contentente un array di byte generato randomicamente ad ogni cerimonia per evitare replay attacks, deve essere lungo almeno 16 Byte e contenere abbastanza entropia da rendere impossibile indovinare il proprio valore;
- **timeout**, parametro facoltativo che identifica in numero i millisecondi che il chiamante aspetta per il completamento dell'operazione, il browser potrebbe sovrascrivere questo valore;

[15] Formato standard per certificati che attestano l'associazione tra una chiave pubblica ed un soggetto.

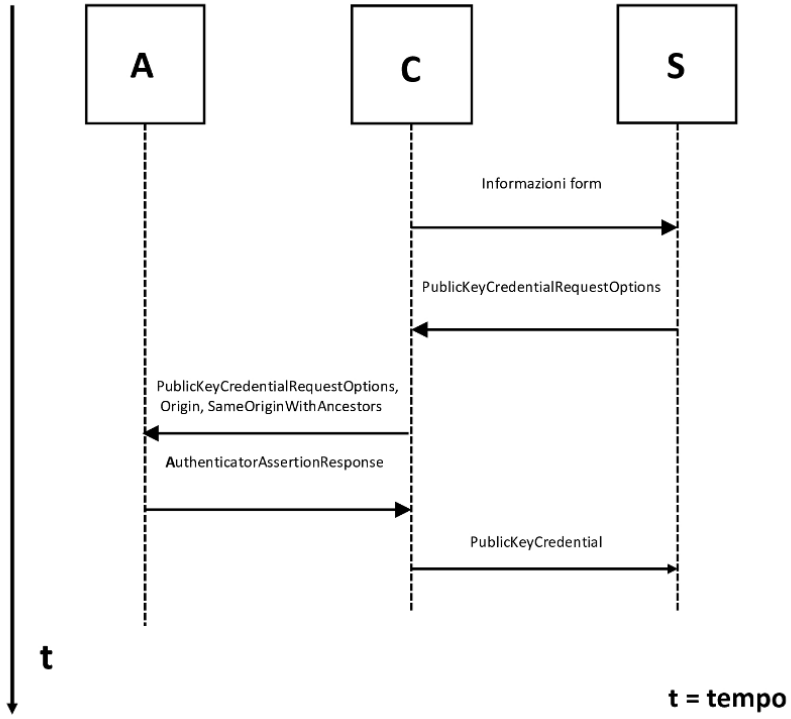

Figura 3.4: *Illustrazione di una cerimonia di autenticazione. A=autenticatore, C=client, S=server*

- **rpId**, parametro opzionale che specifica l'ID del RP, se assente allora il browser imposterà questo parametro con valore pari al dominio del Relying Party;
- **allowCredentials**, parametro opzionale avente come valore un array dove ogni elemento al suo interno descrive una credenziale creata in passato dall'utente ed associata al sito web su cui si sta effettuando la cerimonia di autenticazione. In particolare, all'interno di questo array è contenuto l'ID della credenziale ed un suggerimento su come il client deve comunicare con l'autenticatore per poterla utilizzare. Se omesso, questo parametro avrà come valore un array vuoto.
  N.B.: Se questo parametro dovesse risultare vuoto, non è necessariamente vero che non ci siano credenziali associate al sito web su cui l'utente tenta di accedere, l'utilizzo di *Client-side Credential* implica che l'array sia sempre vuoto nonostante esistano delle credenziali associate al Relying Party;
- **userVerification**, stringa facoltativa posta di default a "*preferred*", indica il requisito del RP di richiedere o meno la verifica dell'utente

da parte dell'autenticatore. Se non richiesta, l'autenticatore richiederà solo un test di presenza, può assumere valori "*required*", "*preferred*" o "*discouraged*".

- **extensions**, parametro opzionale contenente ulteriori elaborazioni richieste per il client o per l'autenticatore, sono definite in "*WebAuthn Extension Identifiers*" situato in [13];

```
publicKeyCredentialRequestOptions = {
  //RANDOM BINARY CHALLENGE
  challenge = "eC8H5nmolg-bpr3fkVq-0JrKNMDDuwE3q1i1k1KE8",
  timeout: 300000,
  rpId: "https://esempio.it",
  allowCredentials =  [
    type: "public-key",
    //CREDENTIAL ID
    id:  "3f0_EHd3rkWe4_LALY_LIcLy2mInAxi42A7wA7GUgmKZLVs",
    transports: ["usb","nfc","ble","internal"]
  ],
  userVerification : "preferred"
}
```

Codice 3.5: *Esempio di publicKeyCredentialRequestOption*

Una volta creato *publicKeyCredentialRequestOptions* (vedi Codice 3.5):

1. Il server ritorna questo oggetto al client;
2. Il browser invoca il metodo *navigator.credentials.get()*, passando come parametri l'oggetto ottenuto nel punto 1), *origin*, tupla allegata dal browser ed identificante l'indirizzo del sito web su cui l'utente vuole effettuare l'autenticazione e *sameOriginWithAncestors*, parametro booleano il quale assume valore *true* se e solo se *origin* coincide con l'indirizzo del sito web che permette la visione del form di autenticazione (si ipotizza la presenza di un iframe all'interno di un sito web A che permette la visione di una pagina appartenente ad un altro sito web, B), vedi paragrafo 4.3.2;
3. Il metodo invocato nel punto precedente controlla ed elabora quanto ricevuto, soprattutto confronta se *origin* sia uguale a *PublicKeyCredentialRequestOptions.rpId*, in caso negativo è un caso di *phishing*, vedi paragrafo 4.3.1;
4. Il client cerca e si connette con l'autenticatore;
5. All'interno del metodo richiamato nel punto 2), il browser ne invoca un altro facente parte del protocollo CTAP2, *authenticatorGetAssertion*, il

quale permette di far ricevere all'autenticatore alcuni dei dati presenti in *publicKeyCredentialRequestOptions*, come l'ID del RP, il parametro *allowCredentials*, un valore booleano indicante *userVerification* o se c'è da compiere un test di presenza, le estensioni se presenti, un oggetto, creato nel punto 3) di questo elenco, il quale prende il nome di *clientDataJSON*, e l'hash in SHA-256 di quest'ultimo.

6. L'autenticatore esegue una verifica dell'utente o un test di presenza a seconda di quanto suggerito dal Relying Party nell'oggetto *publicKeyCredentialRequestOptions* (vedi Codice 3.5 e Figura 3.5);

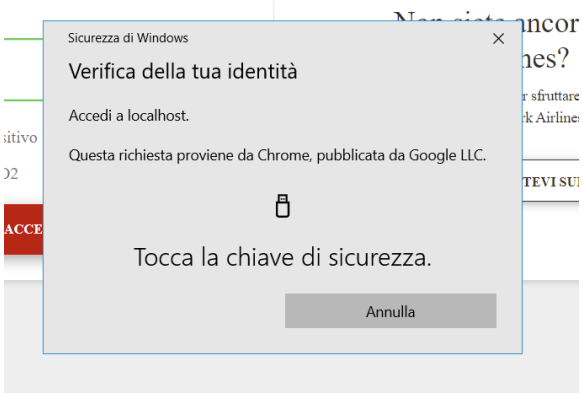

Figura 3.5: *Esempio di test di presenza durante un'autenticazione.*

7. L'autenticatore deve ora generare un'asserzione, per fare ciò si ha bisogno di una *Public Key Credential Source* valida così da ottenere la chiave privata contenuta al suo interno.

   - Se si è nel caso in cui si stanno utilizzando *Server-side Credentials* e il parametro *allowCredentials* è vuoto, allora l'utente non ha credenziali con cui effettuare l'accesso;
   - Se si è nel caso in cui si stanno utilizzando *Server-side Credentials* e l'array contenuto nel parametro *allowCredentials* non è vuoto, allora ogni ID di credenziale viene decriptato (si ricorda che l'ID di una credenziale solo in questo caso è la cifratura di *Public Key Credential Source* e solo l'autenticatore riesce a decifrarlo correttamente, vedi paragrafo 3.1).
     Se dalla decriptazione si ottiene una *Public Key Credential Source* valida allora al suo interno è presente una chiave privata con cui firmare l'asserzione da generare;
   - Se si è nel caso in cui si stanno utilizzando *Client-side Credentials*, attraverso la mappa presente all'interno dell'autenticatore vengono cercate *Public Key Credential Source* con *rpId* validi e

se all'interno dell'autenticatore non ce ne dovessero essere, allora l'utente non ha credenziali con cui effettuare l'accesso;

- Se si è nel caso in cui si stanno utilizzando *Client-side Credentials*, attraverso la mappa presente all'interno dell'autenticatore vengono cercate *Public Key Credential Source* con *rpId* validi e se all'interno dell'autenticatore ce ne dovesse essere salvata almeno una valida, allora nel nell'autenticatore, se questo dispone di uno schermo, o nel browser viene mostrato a video un pop-up in cui è presente una lista di credenziali registrate e associate all'ID del RP, è quindi compito dell'utente sceglierne una con cui effettuare l'accesso, generando così un'asserzione;

8. Nel caso in cui si sia ottenuta un'asserzione e non un errore, questa viene tornata al client sotto forma di struttura contenente diverse informazioni utili. Questa viene elaborata e serializzata dal browser, ottenendo un oggetto, rappresentante la credenziale, che rispetta l'interfaccia *PublicKeyCredential*, il quale al suo interno ha i seguenti parametri:

    - **id**, identificativo della credenziale con cui si sta effettuando l'accesso sotto forma di stringa;
    - **rawId**, identificativo della credenziale con cui si sta effettuando l'accesso ma scritto in forma binaria;
    - **response**, oggetto rispettante l'interfaccia *AuthenticatorAssertionResponse*, contenente:
        - **authenticatorData**, array binario generato dall'autenticatore, simile ad *authData* presente all'interno di *attestationObject* durante la cerimonia di registrazione, ma senza la presenza della chiave pubblica della credenziale al suo interno;
        - **clientDataJSON**, parametro di tipo ArrayBuffer e serializzato in JSON, contiene alcune informazioni che il browser ha passato all'autenticatore, vedi Codice 3.3 e definizione dello stesso campo nella cerimonia di registrazione;
        - **signature**, oggetto di tipo ArrayBuffer generato dall'autenticatore e contenente una firma effettuata da esso concatenando il parametro *authenticatorData* con l'hash di *clientDataJSON*, cifrando il tutto con la chiave privata della credenziale usata per creare l'asserzione;
        - **userHandle**, identificativo dell'utente ritornato dall'autenticatore, può avere valore NULL a causa del fatto che non tutti i dispositivi supportano la memorizzazione di questo dato;
    - **type**, stringa che specifica il tipo di credenziale con cui si sta effettuando l'autenticazione;

- **extensions**, mappa facoltativa i cui elementi al suo interno hanno forma *identificativo delle estensioni* $\mapsto$ *output generato*;

```
PublicKeyCredential = {
  //CREDENTIAL ID
  id:"3f0_EHd3rkWe4_LALY_LIcLy2mInAxi42A7wA7GUgmKZLVs",
  rawId: ArrayBuffer(59),
  response: AuthenticatorAssertionResponse {
    authenticatorData: ArrayBuffer(189),
    clientDataJSON: ArrayBuffer(138),
    signature: ArrayBuffer(65),
    userHandle: ArrayBuffer(10),
  },
  type: 'public-key'
}
```

Codice 3.6: *Esempio di asserzione fornita al server per validazione*

Una volta che il client riceve l'asserzione, questa viene mandata da esso al server per validazione, così da confermare che l'identità dell'utente che vuole effettuare la cerimonia di autenticazione sia veritiera. Dall'altro lato, il server, come per una procedura di registrazione, ha una serie di operazioni da effettuare atte a confermare l'integrità di quanto ricevuto, facendo quindi controlli sul contenuto di PublicKeyCredential in modo da poter autenticare l'utente in tutta sicurezza. Il Relying Party deve effettuare questi controlli al fine di rendere la cerimonia di autenticazione consistente.
Una volta concluso il processo di controllo, il server può dare il via libera e l'utente può finalmente accedere al proprio account, essendo sicuri della sua identità.

3.4 Problematiche riscontrate

3.4.1 Perdita di un autenticatore

In un'autenticazione con password, la chiave per poter accedere ad un servizio web protetto è appartenente alla categoria "*Something Known*"[16], la dimenticanza di tale informazione quindi potrebbe ad esempio impedire l'utente di accedere in un'area riservata, potremmo considerare la password come l'anello debole di tale tipologia d'autenticazione.
Con FIDO2, questa debolezza, trattandosi essenzialmente di cerimonie passwordless, viene trasposta dalla password all'autenticatore.
Con la perdita di un autenticatore, l'utente che vuole effettuare un'autenticazione, non riuscirà a compiere tale azione dato che la credenziale associata

[16] Un utente può provare la propria identità in base ad un'informazione che conosce.

Browser	Supportato	Versione	Data di rilascio
Chrome	✓	67	29 Maggio 2018
Firefox	✓	60	9 Maggio 2019
Safari	✓	13	19 Settembre 2019
Internet Explorer	✗	-	-
Edge	✓	18	13 Novembre 2018
Opera	✓	54	28 Giugno 2018
Safari (iOS)	✓	14.5	26 Aprile 2021
Android Browser	✓	67	29 Maggio 2018
Chrome (Android)	✓	67	29 Maggio 2018
Firefox (Android)	✓	68	5 Agosto 2019
Samsung Internet	✗	-	-
Opera Mobile	✗	-	-

Tabella 3.1: *Compatibilità di WebAuthn API con i principali browser nel mercato.*

al proprio account è salvata all'interno di esso.
Per questo motivo, Fido Alliance e W3C suggeriscono ai RP di permettere il collegamento di molteplici autenticatori ad un account, cosicché nel caso in cui uno di questi venga perso da un utente, ce ne sia almeno uno di scorta con cui continuare ad accedere al servizio offerto.
Quanto scritto è indubbiamente un grande problema per questo sistema di autenticazione, l'avere più autenticatori ha un prezzo in denaro non indifferente da sostenere ed è tutto a carico dell'utente che vuole usufruire di tale tecnologia.

3.4.2 Compatibilità con i browser

WebAuthn API e più in generale FIDO2 stanno prendendo piede molto lentamente all'interno di internet, non molti siti web approfittano di questa tecnologia e quelli che lo fanno, la solamente come secondo fattore di sicurezza in una 2FA.
Questo è causato anche in parte da una non completa compatibilità con i browser presenti nel mercato, come si può vedere nella Tabella 3.1 creata da un'elaborazione di quanto ricavato in [2].

Viceversa, come si può vedere nella Tabella 3.2, creata attraverso un campionamento di 450 siti web randomici presenti in [11] e che meglio si prestano ad un utilizzo del protocollo, la situazione di applicazione di FIDO2 da parte di questi è critica, nota di merito va a Office 365[17] e GitHub[18], gli unici siti

[17] Piattaforma gestita da Microsoft che offre servizi in ambito di produttività.
[18] Piattaforma che permette lo storage e la gestione di progetti software da parte di più persone.

Categoria	N. Siti web analizzati	Num. FIDO2
Bancario	100>	0
Trasporti	40	0
Email	36	0
Governativo	16	0
Vendita	65	0
Social	42	0
Sicurezza	35	0
Intrattenimento	28	0
Altro	~80	2

Tabella 3.2: *Numero di siti web divisi per categoria che usano FIDO2.*

web ad usufruire di tale tecnologia tra i diversi domini presi in considerazione (vedi Figura 3.6).

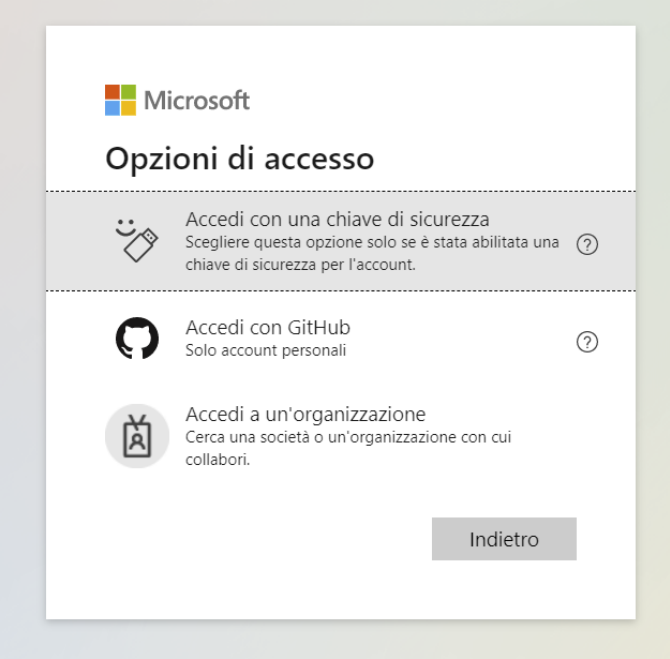

Figura 3.6: *Interfaccia di Office 365 in cui è permesso all'utente di accedere tramite chiave di sicurezza.*

Capitolo 4

Sicurezza e privacy di WebAuthn API

4.1 Threat model

Attraverso un'attenta analisi della documentazione e dell'implementazione di WebAuthn API in applicativi locali, sono state riscontrate diverse possibili vulnerabilità usufruibili da un attaccante; alcune di queste vengono mitigate di base dalla libreria, altre invece si possono risolvere seguendo delle best practices, integrandole correttamente con l'API.
Vengono quindi identificati i possibili threat model, classificandoli come segue:

- **Network Attacker**, si definisce tale una persona che controlla tutti i dati transitanti nella rete internet in cui è situato, con il fine di ricavare informazioni (attacco passivo) e/o danneggiare altre persone (attacco attivo);
- **Web Attacker**, si definisce tale una persona che sfrutta delle vulnerabilità esistenti all'interno di un'applicazione web o browser al fine di ricavare informazioni e/o danneggiare altre persone;

Vengono ora analizzati singolarmente i possibili threat che questi possono attuare.

4.2 Threat attuabili da un network attacker

4.2.1 Man-in-the-middle

La prima vulnerabilità facente parte di questa categoria è il cosiddetto *man-in-the-middle*.
Un network attacker può avvalersi di questa quando ha l'intenzione di intercettare il traffico che avviene tra un computer A ed un computer B, in modo

passivo, quindi solo restando in ascolto nel canale di comunicazione, oppure in modo attivo, cioè decidendo di cambiare a proprio piacere i messaggi che A manda a B e viceversa.

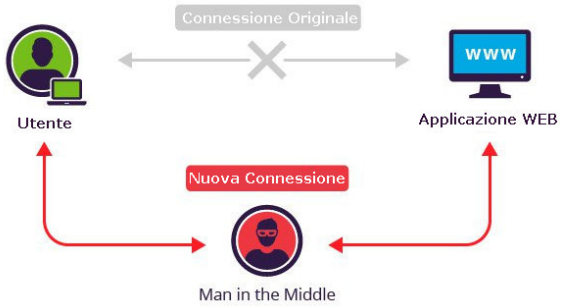

Figura 4.1: *Illustrazione di man-in-the-middle.[14]*

Un canale protetto su cui le informazioni viaggiano in forma criptata può evitare che un malintenzionato riesca a comprendere il testo di un messaggio nonostante riesca ad intercettarlo, ma questo non basta, attaccanti audaci possono, ad esempio, attuare un *downgrade attack*.[1]
Nel caso di WebAuthn API, per cercare di contrastare almeno in parte questa vulnerabilità, nelle sue linee guida, come viene ricordato, è sempre stato esplicitamente richiesto che l'utilizzo della libreria venga effettuato su ambienti protetti da crittografia o in locale nella propria macchina.
Ci si concentra ora sul processo di creazione di credenziali che solitamente un utente compie all'interno di una registrazione ad un servizio, soprattutto in quella fase di essa dove l'autenticatore crea una risposta che fornisce al client e che esso manderà al server per poterlo convalidare.
Viene ricordato, in questa fase un autenticatore ricevendo in input l'oggetto *PublicKeyCredentialCreationOptions* (vedi Codice 3.1), costruisce una risposta che viene fornita al browser e questo, serializzandola, modella *PublicKeyCredential*, che, tra le tante cose, ha al suo interno la chiave pubblica della credenziale appena generata (vedi Codice 3.2).
Questa, si ricorda, viene usata durante il processo di autenticazione dal server per validare la firma che l'autenticatore farà attraverso l'utilizzo della chiave privata associata ad essa, così da permettere il login dell'utente nel servizio richiesto.
Un attaccante, quindi, è interessato a cambiare questa chiave pubblica con una di sua conoscenza, cosicché quando un utente cerca di autenticarsi ad un sito web, la chiave pubblica che esso manda viene sostituita dall'attaccante che è in ascolto e quest'ultimo si autentica al posto dell'utente originale che

[1] Attacco dove viene cambiato, quando possibile, il protocollo di comunicazione tra due computer, passando da una versione sicura di esso ad una meno affidabile o addirittura con la comunicazione in chiaro, vedi [6].

riceve invece un errore.
Si noti quindi che un account può essere violato attraverso questo tipo di attacco se e solo se la creazione di una credenziale è stata compromessa; è importante che l'attaccante rimpiazzi la chiave pubblica creata dall'autenticatore della vittima, quindi, se questo processo dovesse invece risultare sicuro, WebAuthn API sarebbe robusto a questa vulnerabilità anche durante la fase di autenticazione.
Il *man-in-the-middle* può essere facilmente individuato poiché l'utente originale, che voleva autenticarsi al servizio, riceverà degli errori durante i processi di autenticazione, cosa che non dovrebbe accadere se la registrazione fosse andata a buon fine.
Se questo dovesse succedere in maniera continuativa allora l'utente potrebbe avvisare il possessore del sito web, intervenendo su quanto sta accadendo.
Concludendo e ribadendo, quindi, questa vulnerabilità viene limitata, cioè un attaccante non riesce ad impossessarsi dell'account di una persona nonostante legga il traffico del canale, "by design" dall'utilizzo di crittografia asimmetrica e solo nel caso in cui la registrazione non sia stata compromessa, mentre, il requisito che il protocollo funzioni solo su canali di comunicazione sicuri previene in ogni caso questo tipo di vulnerabilità, dunque una sorta di panacea.

4.2.2 Replay attack

Strettamente collegato alla vulnerabilità spiegata nel precedente paragrafo poichè viene richiesta la sua attuabilità, questo attacco mira a replicare un'azione fatta da un client A rispetto ad un server B da parte di un network attacker, così da far credere al server B che la nuova azione sia stata fatta nuovamente dal client A, essendo uguale a quelle fatte in precedenza da esso, quando in realtà dietro le nuove richieste c'è un diverso client, chiamato C, l'attaccante.

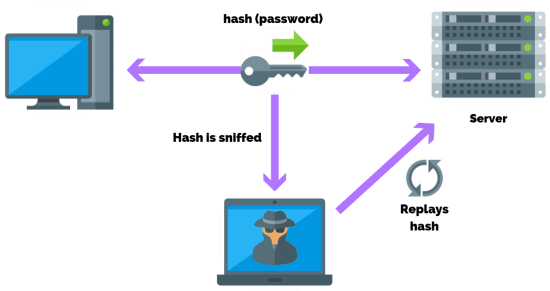

Figura 4.2: *Illustrazione di replay attack.[10]*

Questo attacco, si ribadisce, prevede quindi che l'attaccante sia in grado di ascoltare la comunicazione che avviene tra i due dispositivi A e B, quindi

che il man-in-the-middle sia una vulnerabilità presente ed attuabile.
WebAuthn API ha mitigato, come detto in precedenza, il problema attraverso la richiesta di implementazione della libreria solo in siti web che usufruiscono di protocolli sicuri, come detto per man-in-the-middle.
Inoltre, nell'intero processo di creazione o utilizzo di credenziali, come già descritto in precedenza, all'interno degli oggetti scambiati è presente il parametro **challenge**, il quale contiene un array di byte pseudocasuale che viene creato dal RP ad ogni nuova cerimonia e verrà confrontato al termine di essa, se i due valori non dovessero coincidere allora l'intero processo terminerà in errore.
Questo mitiga "by design" l'attacco perché se un attaccante ha ascoltato un'autenticazione precedente e vuole replicarla, l'attributo *challenge* contenuto nella risposta fornita non sarà valido perché cambiato e l'autenticazione fallirà.
Oltre all'attributo *challenge*, quando supportato dall'autenticatore, è presente un altro sistema di difesa per il processo di autenticazione, questo è l'attributo **signCount**, il quale, viene ricordato, rappresenta il numero di autenticazioni effettuate con una precisa credenziale.
Il suo funzionamento segue questo percorso:

1. Al completamento del processo di registrazione viene salvato all'interno dell'autenticatore e nel server l'attributo *signCount*, assumendo valore 0 ed associandolo alla credenziale a cui fa riferimento;

2. Durante un processo di autenticazione, l'autenticatore crea una risposta da far validare al server, all'interno di questa è presente il valore di *signCount* che era memorizzato al suo interno e che considera anche l'autenticazione in corso;

3. Il server riceve la risposta creata dall'autenticatore e mandata dal client; nel validarla controlla che il valore di *signCount* sia compatibile con quanto memorizzato da se stesso (es. il valore mandato deve essere strettamente maggiore rispetto a quanto salvato nel server).

4. Se la validazione è andata a buon fine, il server aggiorna il proprio valore memorizzato di *signCount* con quanto passato dal client.

Questo può fermare un *replay attack* perché se un attaccante copia ed inoltra al server un messaggio inviato in precedenza da un utente ignaro, il valore di *signCount* salvato nel server sarà uguale o minore rispetto a quello mandato dall'attaccante, di conseguenza l'attacco fallirà.
Si può quindi affermare che WebAuthn API sia resistente a questo tipo di minaccia sia per i requisiti che la libreria pone, cioè quello di funzionare in ambienti sicuri, che per come è costruita, prevedendo delle difese contenute nelle informazioni scambiate tra autenticatore, client e server.

4.3 Threat attuabili da un web attacker

4.3.1 Phishing

Il phishing è uno degli attacchi più comuni in ambito informatico[2] e si basa su tecniche di social engineering[3], contrariamente agli attacchi elencati in precedenza dove si sfruttano debolezze di protocolli o sistemi informatici.
In questo caso si sfruttano vulnerabilità umane, nello specifico si vuole indurre (attraverso email, SMS o altro) un utente che vuole usufruire di un sito web posseduto da A, ad immettere propri dati sensibili in un altro sito web, in tutto e per tutto uguale a quello di A, ma posseduto da B, l'attaccante.

Figura 4.3: *Illustrazione di Phishing.[10]*

WebAuthn API previene che questo attacco accada poiché quando un utente vuole accedere ad un sito web ed il client invoca il metodo *navigator.credentials.get()*, a quest'ultimo viene passato dentro *PublicKeyCredentialRequestOptions*, il quale contiene l'ID del Relying Party fornito dal server, eventualmente malevolo, ed a parte il browser fornisce *origin*, l'indirizzo del sito web su cui l'utente vuole effettuare l'autenticazione; nel caso in cui *PublicKeyCredentialRequestOptions.rpID* e *origin* siano diversi allora si è in un caso di phishing e la libreria fa fallire l'intera cerimonia di autenticazione.
Inoltre, viene ricordato, all'interno dell'autenticatore è presente una mappa che associa *(rpId, [userHandle])* $\mapsto$ *PublicKey Credential Source* nel caso di Client-side Credentials e, viceversa, nel caso di Server-side Credentials l'ID della credenziale è la cifratura tramite crittografia simmetrica di *PublicKey Credential Source* e, per entrambi, prima di creare un'asserzione viene confrontato l'ID del Relying Party contenuto all'interno di questo oggetto e solo

[2] Vedi [8]
[3] Studio del comportamento e della personalità di una persona al fine di ottenere informazioni utili. In informatica può venire utilizzato da malintenzionati in quanto viene spronato un utente ignaro a fornire tali informazioni volontariamente.

in caso sia uguale a quello passato nella richiesta, la credenziale viene considerata valida per l'autenticazione.
Si può quindi affermare che WebAuthn API sia resistente al *phishing* "by design".

4.3.2 Clickjacking

Il *clickjacking* o *UI redressing* è uno degli attacchi più comuni nel web insieme al prima elencato *phishing*.
Il suo funzionamento si basa nell'ingannare un utente nel fargli premere elementi presenti in una pagina web che apparentemente sembra legittima.
In dettaglio:

- Sia A il sito web apparentemente non maligno di un attaccante, quindi con uno stile grafico non ingannevole rispetto al fine mostrato;
- Sia B un sito web posseduto da un ente/società/azienda, B può avere scopi leciti come non;
- Sia B incorporato in A tramite <iframe> posto invisibile, con gli elementi di A allineati a quelli di B e con B posto superiormente ad A;

Un utente utilizzatore di A, intento a premere un elemento al suo interno, premerà invece un elemento di B, posto graficamente superiormente ad A ma reso dall'attaccante volontariamente non visibile.

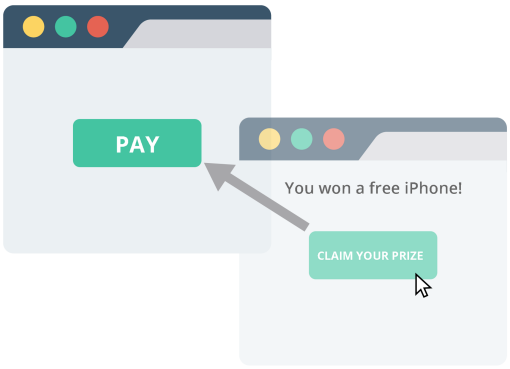

Figura 4.4: *Illustrazione artistica di clickjacking.[5]*

Alcuni utilizzi pratici di utilizzo di questo attacco sono:

- Far acquistare un oggetto/servizio ad un utente ignaro;

- Far interagire l'utente ignaro con persone/pagine/post presenti in social network[4];
- Effettuare transazioni di denaro non previste dall'utente ignaro;

WebAuthn API di default disabilita la possibilità di utilizzo della libreria in <iframe> situati in domini diversi da quello originale e soprattutto vieta la possibilità di creare credenziali attraverso questi, ciò nonostante è possibile rilassare il vincolo riguardante l'utilizzo di credenziali già esistenti[5], ipoteticamente quindi sarebbe possibile effettuare processi di login o operazioni in cui WebAuthn API è utilizzata come secondo livello di sicurezza in una 2FA. Fare ciò renderebbe possibile il *Clickjacking* ed è compito del Relying Party assicurarsi che l'iframe sia visibile nel sito esterno, un esempio è usando l'API Intersection Observer [6].

4.3.3 Cross-site Scripting

Il *Cross-site Scripting* (XSS) è un tipo di attacco in cui un ipotetico attaccante inietta del codice dannoso all'interno del browser di una persona al fine di, tra le varie motivazioni, rubare informazioni memorizzate all'interno dell'applicazione (es. cookies, localstorage, ecc...), modificare il comportamento delle pagine web visualizzate o addirittura fungere da keylogger[7].
Questo attacco può essere identificato in 4 varianti, queste identificate dall'intersezione degli elementi facenti parte nelle seguenti 2 macro-categorie:

- Tipo di vulnerabilità:
  - *Reflected XSS*, attacco in cui l'applicazione web "riflette" del codice malevolo dato in input al browser, prendendo di mira utenti specifici che visitano link dannosi;
  - *Stored XSS*, attacco in cui l'applicazione web memorizza al proprio interno il codice malevolo iniettato e lo "serve" successivamente alle persone visitanti le pagine della stessa applicazione web;

[4] Piattaforma web che permette l'interazione tra persone, permettendo tra di loro la condivisione di testo, immagini, video, ecc...

[5] Attraverso le "*Feature Policy*" è possibile abilitare, disabilitare o modificare l'utilizzo o il comportamento di funzionalità ed API utilizzate.
Per poter utilizzare le "*Feature Policy*" ci sono due modi:

- Attraverso l'HTTP header Permissions-Policy;
- Attraverso l'attributo "*allow*" in un tag <iframe>.

[6] Libreria che permette di capire la visibilità e la posizione di elementi in una pagina HTML.

[7] Strumento utile alla cattura di ogni singolo tasto premuto sulla tastiera di un device.

- Luogo in cui è presente la vulnerabilità:
  - *Server-side XSS*, l'attacco sfrutta il server dell'applicazione web legittima per avere atto;
  - *Client-side XSS o DOM-based XSS*, l'attacco avviene esclusivamente attraverso il browser della vittima e non prevede l'utilizzo del server dell'applicazione web coinvolta;

Il Cross-site Scripting non è legato in alcun modo all'architettura del protocollo FIDO2 che, suo malgrado, non riuscirebbe a distinguere le azioni fatte da un utente piuttosto che dal codice dannoso iniettato da un attaccante e di conseguenza il protocollo risulta vulnerabile a questo genere di attacco; parallelamente va sottolineato che l'interazione umana richiesta in FIDO2 durante un processo di autenticazione potrebbe minare la riuscita dell'attacco stesso.
Viene presentata ora un'ipotesi di Reflected Server-side XSS con FIDO2.
Assumendo:

1. La presenza di un sito web S adibito all'invio ed alla ricezione di denaro;
2. La vittima dell'attacco ha effettuato un login all'interno di S e la sessione è valida nel momento in cui accade l'exploit;
3. S prevede l'utilizzo di FIDO2 come sistema di autenticazione prima di ogni invio di denaro ad una persona P;

Un esempio di attacco può essere riassunto nei seguenti passi:

1. Un attaccante viola il browser di una vittima iniettando del codice malevolo al suo interno (per esempio inviando alla vittima un link contenente lo script), questo codice prevede l'invio di richieste HTTP ad S, in cui l'utente ha precedentemente fatto l'accesso; il fatto che queste richieste siano generate da un browser ed una sessione lecite non desta sospetti al RP, bypassando di fatto la *Same Origin Policy* (SOP)[8];
2. Il codice malevolo crea una richiesta HTTP indirizzata al Relying Party chiedendo di iniziare una cerimonia di autenticazione per l'invio di una somma di denaro X ad una persona P;
3. Il RP manda al browser un oggetto publicKeyCredentialRequestOptions, contenente tra le tante informazioni il campo "userVerification" (vedi 3.3.2);

[8] Meccanismi di difesa applicati ad una risorsa al fine di legittimare la lettura e scrittura su questa solamente a chi la controlla e non a script proveniente da fonti terze.

4. Nel caso in cui sia solo richiesto un test di presenza, un utente disattento o tratto in inganno potrebbe confermare la transazione;

5. La cerimonia continua ed al suo termine l'invio di denaro è compiuto e P riceve X;

Come si può notare, il punto 4 dell'elenco immediatamente sopra è il più delicato e prevede l'interazione dell'utente con l'autenticatore e ciò è necessario per la riuscita o meno dell'exploit, quindi una verifica dell'utente più solida (es. userVerification required) potrebbe prevenire in maniera più consistente la realizzazione dell'attacco che, comunque può avere luogo e spetta quindi agli sviluppatori delle applicazioni web rendere sicure le proprie piattaforme utilizzanti FIDO2.

4.3.4 Privacy leak

Username/email leak attraverso un processo di registrazione

Un attaccante potrebbe avere come scopo quello di sapere se un utente sia registrato o meno ad un sito web compilando il form di registrazione dello stesso sito web attaccato con l'username/email della persona attaccata.
Al fine di avere un'esperienza ancora più sicura, le linee guida suggerite da WebAuthn suggeriscono di non identificare un utente tramite email quando possibile, ma solo attraverso uno username, questo perché una persona può usare username diversi in siti web diversi, questo ridurrebbe l'impatto che si ha nel momento in cui un attaccante viene a scoprirlo, poiché in tal caso uno username non provocherebbe la deanonimizzazione che un'email causerebbe.
Nel caso in cui l'utilizzo di email sia inevitabile al fine di identificare un utente, si consiglia il seguente approccio:

- Quando un utente fornisce la propria email durante un processo di registrazione, fermare tale procedimento ed inviare un codice OTP[9] all'indirizzo inserito;

- Mostrare a video gli stessi messaggi indipendentemente dal fatto che la email sia o non sia già registrata al sito web.

Oltre a quanto elencato, è possibile scoprire se una persona sia registrata o meno ad una piattaforma web analizzando le risposte che il server fornisce al client, in questo caso l'attaccante, in particolare, all'interno di *publicKeyCredentialCreationOptions* è contenuto l'attributo **excludeCredentials** (vedi Codice 3.1), ricordando il suo significato:

- Se l'array è vuoto, l'account con username immesso non ha credenziali associate ad esso;

[9] One-time Password, password che valida solo una precisa transazione in un preciso intervallo di tempo.

- Se l'array non è vuoto, l'account con username immesso ha almeno una credenziale associata ad esso;

Attraverso la visione di questo attributo, un attaccante è quindi in grado di determinare la presenza di un account che abbia effettuato il processo di registrazione attraverso questa API.

Username/email leak attraverso un processo di autenticazione

Similmente a quanto accade per la registrazione, un attaccante può voler sapere se un utente sia registrato o meno ad un sito web compilando il form di autenticazione dello stesso sito web attaccato e con lo username/email della persona interessata.
Questo è scopribile attraverso la visione dell'array **allowCredentials** facente parte di *publicKeyCredentialRequestOptions* (vedi Codice 3.5), che si ricorda essere un oggetto contenuto nella risposta che il server restituisce al client nel momento in cui l'utente fa capire al Relying Party di volere iniziare una cerimonia di login.

N.B.: Questa vulnerabilità è possibile solamente nel caso in cui si è in presenza di *Server-side Credential*, poiché solo in questo caso l'array *allowCredentials* potrebbe contenere degli ID di credenziali.

Nel caso di utilizzo di *Server-side Credential*, le casistiche sono:

- *allowCredentials* è vuoto, quindi non ci sono credenziali associate allo username inserito;
- *allowCredentials* non è vuoto, quindi ci sono una o più credenziali associate allo username inserito;

Per contrastare questa divulgazione di informazioni, il Relying Party potrebbe creare dei dati fittizi e sintatticamente validi con cui popolare *allowCredentials* e da usare nel caso in cui l'utente, che l'attaccante cerca, non sia registrato.
Così facendo, nella situazione in cui un attaccante immette nel form di autenticazione uno username/email altrui, egli si vede ricevere dei dati che sembrano corretti sia che l'utente sia registrato, sia nel caso contrario e quindi non riesce a capire in quali dei due casi si trovi.
È buona prassi variare ad ogni richiesta questi dati fittizi creati, altrimenti con username diversi, se l'attaccante dovesse ricevere sempre la stessa risposta potrebbe intuire la casistica.

Credential ID leak

Un Credential ID leak, diversamente dai casi precedenti, non consente all'attaccante di scoprire in maniera diretta informazioni personali di un utente,

ciò nonostante, è altrettanto pericoloso per la sicurezza dell'utente coinvolto.

N.B.: Questa vulnerabilità è possibile solamente nel caso in cui si è in presenza di *Server-side Credential*, poiché solo in questo caso l'array *allowCredentials* potrebbe contenere degli ID di credenziali.

Ricordando la struttura di *publicKeyCredentialRequestOptions* (vedi Codice 3.5), il parametro *allowCredentials* contiene tutti gli ID delle credenziali collegate all'account dell'utente con cui questo sta effettuando l'autenticazione. Tale oggetto viene fornito in input all'autenticatore dell'utente, il quale ne restituisce uno nuovo contenente, tra le tante cose, uno degli ID elencati in precedenza e con cui l'utente ha deciso di effettuare l'accesso (vedi Codice 3.6).
Avendo fatto in precedenza diverse considerazioni riguardo l'attacco *man-in-the-middle* e ciò che ne comporta, analogamente, in questo caso un attaccante in grado di leggere in chiaro la comunicazione che avviene tra una client A ed un server B è in grado di leggere tutti gli ID associati ad un preciso utente e può trarre diverse considerazioni da questi, cioè:

- L'attaccante può capire il tipo di autenticatore utilizzato in base alla lunghezza dell'ID;
- È probabile che un utente utilizzi lo stesso username e lo stesso set di autenticatori in più siti web, questo potrebbe aiutare l'attaccante a deanonimizzarlo;

Per evitare che ciò accada, si possono attuare una od entrambe le seguenti soluzioni:

- Usare *Client-side Credential*;
- Effettuare un'autenticazione separata, usando per esempio username e password e solo successivamente avviare l'autenticazione FIDO2 esponendo gli ID delle credenziali.

Scoperta di account con protezione "debole"

Durante il processo di autenticazione, attraverso la compilazione del form di login, un attaccante potrebbe essere interessato a concentrarsi nella ricerca di account con i quali non è prevista un'autenticazione tramite FIDO2, ma invece attraverso una classica autenticazione con username e password.

N.B.: Questa vulnerabilità è possibile solamente nel caso in cui si è in presenza di *Server-side Credential*, poiché solo in questo caso l'array *allowCredentials* potrebbe contenere degli ID di credenziali.

Assumendo che l'aggressore sappia per certo che l'account attaccato sia registrato alla piattaforma presa di mira, similmente a quanto visto nel paragrafo 4.3.4 e 4.3.4, l'attaccante usufruisce delle informazioni fornite dall'array *allowCredentials*, in particolare:

- Se l'array dovesse essere vuoto inserendo lo username dell'account attaccato, allora questo non effettua l'autenticazione tramite FIDO2;
- Se l'array non dovesse essere vuoto inserendo lo username dell'account attaccato, allora questo effettua l'autenticazione tramite FIDO2;

In base a quanto scoperto, una persona malintenzionata potrebbe concentrarsi su tutti gli account con autenticazione non passwordless, attuando tutti gli attacchi che questo tipo di autenticazione soffre.
Per mitigare questa vulnerabilità, il Relying Party prima di fornire l'array, se questo dovesse risultare vuoto, potrebbe decidere di riempirlo con dati fittizi e sintatticamente validi.
Così facendo, nella situazione in cui un attaccante immette nel form di autenticazione uno username/email altrui, egli si vede ricevere dei dati che sembrano corretti sia che l'utente effettui l'autenticazione tramite FIDO2, sia nel caso contrario e quindi non riesce a capire in quali dei due casi si trovi.
È buona prassi variare ad ogni richiesta questi dati fittizi creati, altrimenti con username diversi immessi, se l'attaccante dovesse ricevere sempre la stessa risposta potrebbe intuire la casistica.

Capitolo 5

Analisi formale

Nonostante un'analisi primordiale di sicurezza sia stata fatta nel capitolo precedente, non tutte le vulnerabilità potrebbero essere emerse o, soprattutto, potrebbero essere state fatte deduzioni errate riguardo quelle trovate, questo perché la ricerca si è basata su una lettura della documentazione ufficiale e da quanto riscontrato attraverso l'implementazione dell'API in applicativi di test locali allo scrittore della tesi.
In questo capitolo, quindi, si procede alla verifica della sicurezza del protocollo WebAuthn attraverso l'utilizzo di un nuovo strumento, *ProVerif*[1].
Questo tool permette di stabilire in maniera automatica e matematicamente la solidità di un protocollo opportunamente modellato ed espresso in un linguaggio all'interno di un file, seguendo il modello *Dolev-Yao*[2] e supportando la verifica delle seguenti proprietà:

- **secrecy** se un processo P preserva la riservatezza di M, cioè nel caso in cui nessun attaccante O è in grado di ricostruire M o di vedere lo stesso in chiaro all'interno dello svolgimento di P;
- **authentication** se le occorrenze in esecuzione degli eventi definiti $E_1(n_1,...,n_k)$,$E_2(m_1,...,n_j)$,...,$E_i(l_1,...,l_h)$ con $i,k,j,h \in \mathbb{N}$ rispettano la linea temporale prevista dall'implementazione del protocollo e non si susseguono in un ordine non previsto;
- **strong secrecy** se un attaccante O non è in grado di distinguere due o più sessioni del protocollo in esecuzione nonostante i messaggi che viaggiano siano cambiati tra una sessione e l'altra;

Fornendo poi 3 possibili output una volta eseguito il programma (**N.B.**: potrebbe accadere che l'esecuzione di un programma non termini mai):

[1] Per saperne di più, visitare https://prosecco.gforge.inria.fr/personal/bblanche/proverif/
[2] Modello in cui le primitive crittografiche sono viste come "black box" modellate attraverso funzioni, quindi l'attaccante non può modificare il loro funzionamento ma al massimo vedere e/o usare il risultato fornito da loro in output.

- **safe** se la proprietà analizzata non può essere violata;
- **unsafe** se la proprietà analizzata potrebbe essere violata;
- **unsure** se non è possibile provare la sicurezza o, viceversa, la non sicurezza di una proprietà;

5.1 Applied Pi Calculus

Come accennato all'inizio del capitolo, il protocollo viene formalizzato attraverso un linguaggio, in questo caso è stato utilizzato un dialetto di *Applied Pi Calculus*, linguaggio progettato da M.Abadi e C.Fournet *(Abadi et al., 2001)*[15] e spesso utile a formalizzare protocolli inerenti l'ambito della sicurezza informatica.
Il linguaggio con il quale è stato definito il protocollo viene poi tradotto, al momento dell'esecuzione, in clausole di Horn[3], attraverso le quali l'algoritmo risolutore cercherà di raggiungere gli obiettivi posti simulando un attaccante che opera sul protocollo in esecuzione.
Diamo ora una breve introduzione alla notazione utilizzata per modellare l'API.

5.1.1 Linguaggio

Termini, costruttori e distruttori

I termini rappresentato i dati ed i messaggi transitanti nel protocollo e possono avere un ***tipo*** T (di default già presente o definito manualmente dall'utente tramite sintassi *type nometipo* ad inizio documento), per esempio *channel* per identificare il canale di trasmissione, *bitstring* per le stringhe, etc.
Il linguaggio è sviluppato su un infinito insieme di nomi N rappresentanti dati atomici, variabili V e su un insieme finito F contenente delle funzioni, ognuna con la propria arietà[4] $ar(f)$.
I termini sono costruiti dall'applicazione di una qualsiasi $f \in F$ su nomi $n \in N$, variabili $v \in V$ ed altri termini, in questo caso f prende il nome di ***costruttore*** e viene espresso all'interno del documento su cui viene trascritto il protocollo attraverso la sintassi:

$$\textbf{\textit{fun}}\ f(T_1, T_2, ..., T_{\mathrm{n}}) : T$$

dove f è il nome del costruttore con arietà $n \in \mathbb{N}$, T_1, T_2,..., T_{n} sono i tipi dei parametri forniti alla funzione e T è il tipo del termine tornato in output.

[3] Disgiunzione di letterali in cui al massimo uno di questi è positivo
[4] Numero di argomenti presi in input da una funzione.

Viceversa, per manipolare termini creati da costruttori e ricavare delle informazioni usate per la creazione del termine stesso o semplicemente effettuare operazioni attraverso esso, si fa utilizzo di ***distruttori***, si può quindi evincere la seguente affermazione, cioè che ogni costruttore ha un numero $n \in \mathbb{N}$ di distruttori.
I distruttori vengono definiti all'interno del documento su cui viene formato il protocollo attraverso la sintassi:

$$\textbf{\textit{reduc forall}}\ x_1 : T_1, ..., x_\mathrm{m} : T_\mathrm{m}; h(K_1, ..., K_\mathrm{n}) = M.$$

dove h è il distruttore di arietà $n \in \mathbb{N}$, K_1,...,K_n sono i termini forniti come parametro ad h, venendo questi "costruiti" dall'applicazione di costruttori su x_1:T_1,...,x_m:T_m.
I termini sono quindi formalmente definiti dalla seguente grammatica:

$t_1, t_2, t_3, ... ::=$			termini
	n	$n \in N$	nome
	v	$v \in V$	variabile
	(t_1, t_2, t_3)		tupla
	$f(t_1, ..., t_\mathrm{m})$	$f \in F,\ m = ar(f))$	applicazione di costruttore o distruttore

Espressioni

Un'espressione è un termine o una computazione su uno o più espressioni e può essere formalmente espressa nella seguente grammatica:

$D ::==$		espressioni
	t	termine
	$h(D_1, ..., D_\mathrm{n})$ dove $n \in \mathbb{N}$	applicazione di funzione
	fail	fallimento

Processi

I processi possono essere visti come l'intero programma (protocollo formalizzato) o parti di esso che lo compongono. In seguito la sintassi concessa nell'utilizzo di questi:

$P, Q, R ::==$		processi
	0	processo null
	$P \mid Q$	composizione parallela
	$!P$	replicazione
	$new\ \ v : T; P$	restrizione
	$if\ \ t_1 = t_2\ \ then\ \ P\ \ else\ \ Q$	condizione

$in(t_1, v : T); P$	input
$out(t_1); P$	output
$let\ v : T = D\ in\ P\ else\ Q$	risoluzione di espressione

Il processo 0 non fa nulla, $P|Q$ è la composizione parallela di P e Q, questo è usato per rappresentare il lavorare in parallelo di P e Q, $!P$ è la replicazione del processo P, cioè il moltiplicarsi "illimitato" di sessione in esecuzione del processo P, cioè $P|P|$ ..., mentre $newv : T; P$ lega v a P, rendendolo visibile solo al suo interno. La condizione *if*, invece, valuta il valore di termini o espressioni, così da decidere l'esecuzione di un processo piuttosto che un altro, *input* riceve in ingresso da un altro processo dei termini mentre *ouput* fa l'esatto opposto, mandando dati ad un altro processo, per finire, la *risoluzione di espressione* consente di ottenere un terminale dallo svolgimento di un'espressione.

Eventi

Un evento è un'astrazione di fatti ritenuti importanti all'interno del protocollo, venendo così raggruppati sotto un un nome comune. È bene sottolineare che il raggiungimento di un evento durante l'esecuzione di un protocollo non influenza in alcun modo il comportamento successivo dell'esecuzione stessa. Gli eventi vengono dichiarati nella forma:

$$\textbf{\textit{event}}\ ev(K_1, ..., K_n); \qquad n \in \mathbb{N}$$

dove K_1,...,K_n sono termini.

5.2 Modellando WebAuthn API

Viene presentata ora, dopo una veloce introduzione del dialetto utilizzato, una rappresentazione dell'API attraverso Applied Pi Calculus dove vengono provate le proprietà "Secrecy" e "Authentication" del protocollo, sia in cerimonie di registrazione che di autenticazione.
Per cominciare, vengono elencati i tipi definiti manualmente dall'autore ed assegnati ai termini, le variabili su cui si è interessati a sapere se la propria integrità sia stata compromessa, le primitive crittografiche espresse come costruttori e distruttori.

Tipi

- **challenge**, identifica una stringa pseudocasuale creata all'inizio di ogni cerimonia per evitare replay attacks.
- **attskey**, identifica la chiave privata di attestazione usata dall'auutenticatore per firmare la credenziale creata;

- **attpkey**, identifica la chiave pubblica di attestazione usata per verificare la firma contenuta in attestationObject durante la cerimonia di registrazione;
- **skey**, identifica la chiave privata di una credenziale creata da un autenticatore;
- **pkey**, identifica la chiave pubblica di una credenziale creata da un autenticatore;
- **key**, identifica la chiave usata dall'autenticatore per cifrare e decifrare *Public Key Credential Source*, ottenendo un valore usato come id della credenziale, presente solo in caso di utilizzo di Server-side Credentials;
- **keyChannel**, identifica la chiave usata per criptare il canale di comunicazione usato da client e server per scambiare informazioni, come similmente fatto in *(Guirat et al., 2018)*[17];

Variabili

Ecco un elenco di tutte le variabili utilizzate nella rappresentazione del protocollo, queste, salvo espressa indicazione, sono definite di partenza [*private*], cioè non inizialmente a conoscenza di un attaccante, questo non vieta che il loro valore non possa essere compromesso durante l'esecuzione del protocollo.
Le variabili in seguito elencate rispettano l'ordine di lettura **valore rappresentato dalla variabile** (*variabile : tipo*).

- **Canale di trasmissione** (c : *channel*), è il canale in cui passano le informazioni tra client e server, la sua criptazione viene gestita attraverso una chiave segreta (conosciuta solo dal client e dal server) di tipo *keyChannel* che, utilizzando crittografia simmetrica, simula l'utilizzo di HTTPS su di esso, similmente a quanto fatto in *(Guirat et al., 2018)*[17];
- **Chiave privata di attestazione** (*sakAuth* : *attskey*), utile per firmare le credenziali create dall'autenticatore;
- **Chiave pubblica di attestazione** (*pakAuth* : *attpkey*), utile per poter verificare la firma di attestazione fatta attraverso la corrispondente chiave privata;
- **Chiave privata di una credenziale** (*secretKey* : *skey*), parte segreta della coppia di chiavi generate da un autenticatore per una credenziale;
- **Chiave pubblica di una credenziale** (*publicKey* : *pkey*), parte pubblica della coppia di chiavi generate da un autenticatore per una

credenziale, definita [private] ma potrebbe essere considerata a conoscenza dell'attaccante se si ipotizza un data breach[5] nel luogo in cui il Relying Party memorizza i dati degli utenti registrati, quindi sarebbe resa pubblica solo al termine di una registrazione, questo viene deciso a seconda della filosofia di pensiero ma il risultato finale sull'integrità del protocollo che viene fornito in output dal programma non cambia;

- **ID della credenziale** (*crID* : *bitstring*), stringa pseudocasuale che identifica una credenziale durante una registrazione o autenticazione, settata come [*private*] a seconda che si utilizzino credenziali lato server o client, questo perché nel caso delle prime, per simulare il comportamento spiegato nel paragrafo 4.3.4, si da per scontato che l'attaccante conosca già il valore dell'ID;

- **Chiave di cifratura di un autenticatore** (*keyAuth* : *key*), chiave con cui un autenticatore cifra *Public Key Credential Source* (in questo modello rappresentato dalla tripla contenente la chiave privata di una credenziale, l'ID di un utente e l'ID del Relying Party), in modo da rappresentare l'ID di una credenziale nel caso di utilizzo di Server-side Credentials.
  N.B.: Per ovvi motivi questa variabile è presente solo nel caso di utilizzo di Server-side Credentials;

- **Chiave di cifratura su canale di comunicazione** (*key* : *keyChannel*), chiave con cui il canale di comunicazione tra client e server viene cifrato in modo da simulare l'utilizzo di HTTPS su di esso, questo solo nel caso in cui questa sia settata come [private] (come similmente fatto in *(Guirat et al., 2018)*[17]), al contrario, nel caso non dovesse esserlo, il canale di comunicazione sarà vulnerabile a network attacker data la chiara visiblità delle informazioni transitanti su di esso.

Costruttori

Come già anticipato nel paragrafo 5.1.1, i costruttori sono funzioni usate per creare termini, in particolare questi vengono trattati all'interno di ProVerif per modellare primitive crittografiche.
Viene ora fornita una lista, affiancata da una breve spiegazione di ogni costruttore utilizzato all'interno della formalizzazione di WebAuthn API.

- **fun pk(*skey*):pkey**, dove data una chiave privata, mi viene ritornata la corrispondente chiave pubblica;

[5] Violazione di dati in un sistema informatico.

- **fun sign(*bitstring*, *skey*):bitstring**, dove data una stringa qualsiasi ed una chiave privata di una credenziale, la prima viene firmata attraverso l'utilizzo della seconda;
- **fun signAtt(*bitstring*, *attskey*):bitstring**, dove data una stringa qualsiasi ed una chiave privata di attestazione, la prima viene firmata attraverso l'utilizzo della seconda;
- **fun senc(*skey*, *bitstring*, *bitstring*, *bitstring*, *key*):bitstring**, dove data la chiave privata di una credenziale, l'ID dello username e del Relying Party, il tutto viene criptato ottenendo una stringa identificante l'ID di una credenziale.
  N.B.:Questo costruttore è presente solo in caso di Server-side Credentials;
- **fun sencChannel(*bitstring*, *keyChannel*):bitstring**, dove dato un messaggio qualsiasi ed una chiave di tipo *keyChannel*, il primo parametro viene criptato utilizzando il secondo, ottenendo così un messaggio cifrato (crittografia simmetrica).

Distruttori

Come detto nel paragrafo 5.1.1, i distruttori sono funzioni che operano su termini creati da costruttori, quindi, dato in ingresso al distruttore un termine, esso è in grado di ricavare delle informazioni usate per la creazione del termine stesso o di effettuare operazioni su di esso.
Viene ora fornita una lista affiancata da una breve spiegazione di ogni distruttore utilizzato all'interno della formalizzazione di WebAuthn API.

- **reduc forall m: *bitstring*, kiS:*skey*, kiP:*pkey*; checksign(sign(m,kiS),kiP) = m.** dove, avendo una firma creata utilizzando la chiave privata di una credenziale, controllo che questa firma sia valida attraverso l'utilizzo della corrispondente chiave pubblica appartenente alla stessa credenziale e ritorno il messaggio su cui è stata calcolata;
- **reduc forall m: *bitstring*, ki:*skey*; getmess(sign(m,ki)) = m.** dove, avendo una firma creata utilizzando la chiave privata di una credenziale, ritorno il messaggio su cui è stata calcolata;
- **reduc forall m: *bitstring*, kiS:*attskey*, kiP: *attpkey*; checksignAtt(signAtt(m,kiS),kiP) = m.** dove, avendo una firma creata utilizzando la chiave privata di attestazione di un autenticatore, controllo che questa firma sia valida attraverso l'utilizzo della corrispondente chiave pubblica appartenente allo stesso autenticatore e ritorno il messaggio su cui è stata calcolata;

- **reduc forall m:** ***bitstring*****, ki:*****attskey*****; getmessAtt(signAtt(m,ki)) = m.** dove, avendo una firma creata utilizzando la chiave privata di attestazione di un autenticatore, ritorno il messaggio su cui è stata calcolata;

- **reduc forall sec:** ***skey*****, m1:*****bitstring*****, m2:*****bitstring*****, ki:*****skey*****; sdec(senc(sec,m1,m2,ki),ki) = (sec,m1,m2).** dove, avendo un messaggio cifrato rappresentante un ID di credenziale e creato tramite chiave *ki* appartenente all'autenticatore, viene decifrato ottenendo una tripla rappresentante una chiave privata di una credenziale, l'ID di un utente e del Relying Party.
  N.B.:Questo distruttore è presente solo in caso di Server-side Credentials;

- **reduc forall m:** ***bitstring*****, k:*****keyChannel*****; sdecChannel(sencChannel(m,k),k) = m.** dove, avendo un messaggio criptato creato utilizzando crittografia simmetrica, ritorno il messaggio originale decriptato;

5.2.1 Threat Model

In questa rappresentazione del modello viene simulato un *network attacker* (vedi paragrafo 4.1 e 4.2) che può vedere e modificare il traffico tra client e server all'interno delle due cerimonie.
Il protocollo è stato formalizzato su due distinti scenari, sia come operante su canale privato che su pubblico, così da capire quanto sia importante il requisito che obbliga l'utilizzo di WebAuthn solo su applicativi che comunicano tramite protocollo HTTPS.

5.2.2 Processi

Le entità modellate all'interno del protocollo saranno due (si ricordi che viene formalizzato WebAuthn API e non FIDO2, l'autenticatore quindi non viene riportato ed i risultati da esso creati vengono modellati per semplicità di rappresentazione come se fossero stati creati all'interno del client):

- **client**;

- **server**;

queste saranno rappresentate all'interno della formalizzazione del programma come due processi distinti che comunicheranno tra di loro scambiandosi dati durante entrambe le cerimonie di registrazione e autenticazione.

5.2.3 Cerimonie

Registrazione

La cerimonia di registrazione (vedi paragrafo 3.3.1) può essere separata in 3 fasi, la prima vede l'utente compilare attraverso il client un form di registrazione richiesto per potersi registrare al sito web, la seconda prevede che il server leghi all'utente uno userID e fornisca al client le informazioni utili per poter creare una credenziale tramite un autenticatore posseduto dall'utente, infine, la terza vede il client fornire al server tutto il necessario per validare la credenziale appena creata e informazioni utili da salvare per poter compiere poi delle autenticazioni future. Queste 3 fasi possono essere riassunte tramite la seguente sintassi:

C = client
S = server

1) $C \mapsto S$: info (informazioni fornite nel form)
2) $S \mapsto C$: ($chal$, $rpID$, $userID$)
3) $C \mapsto S$: sign(($publicKey$,$chal$,$pakAuth$,$crID$,$rpID$), $sakAuth$)

Autenticazione

Come per la registrazione, anche l'autenticazione (vedi paragrafo 3.3.2) può essere divisa in 3 fasi, la prima prevede l'interazione dell'utente nel compilare il form di login, facendo conoscere al server l'inizio della cerimonia, la seconda vede il server mandare al client le informazioni necessarie cosicchè questo le fornisca all'autenticatore, in modo che possa trovare la credenziale adatta con cui accedere al servizio e, per finire, la terza vede coinvolti il client che fornisce al server i dati associati alla credenziale con cui accedere e da validare, così da consentire il login. Le 3 fasi possono essere riassunte tramite la seguente sintassi:

C = client
S = server

1) $C \mapsto S$: info (informazioni fornite nel form)
2) $S \mapsto C$: ($chal$, $rpID$, $crID$) (senza $crID$ nel caso di credenziali client-side)
3) $C \mapsto S$: sign(($chal$,$crID$,$rpID$), $secretKey$)

5.2.4 Formalizzazione

Avendo introdotto ProVerif, il dialetto utilizzato per rappresentare l'API, il threat model, le entità coinvolte e una rappresentazione primordiale delle due cerimonie, viene mostrato ora nel lato pratico come tutto ciò sia stato combinato per formalizzare il protocollo analizzato.

Proprietà di segretezza

Per dimostrare la proprietà "*secrecy*" di WebAuthn e modellare così un network attacker, seguendo quanto detto nel paragrafo 5.2.1, sono state definite le seguenti query (vedi Codice 5.1).

```
query attacker(pakAuth).
query attacker(sakAuth).
query attacker(publicKey).
query attacker(secretKey).
query attacker(crID).
query attacker(keyAuth). (*Solo per Server-side Credentials*)
query attacker(k).
```

Codice 5.1: *Query per dimostrare la proprietà secrecy del protocollo*

La sintassi ***query*** *attacker(<message>)* permette di captare la segretezza di un dato termine <message>, simulando un attaccante intento a scovare il contenuto di un'informazione e nel caso in cui l'integrità dell'informazione stessa viene a mancare, l'esecuzione della relativa query ritornerà *false*.
In questo caso, quindi, un attaccante potrebbe essere interessato all'intercettazione di valori considerati di importanza chiave per l'intera cerimonia di registrazione e/o autenticazione durante uno od entrambi questi due processi, ascoltando passivamente o attivamente la comunicazione in corso d'opera (vedi paragrafo 4.2.1), questi sono la chiave pubblica e privata di attestazione associati ad un autenticatore, la chiave pubblica e privata di una credenziale, l'identificativo di una credenziale e la chiave con cui un autenticatore cifra Public Key Credential Source. Per completezza è stata inserita anche la chiave usata nella crittografia simmetrica per simulare il comportamento di un canale di comunicazione sicuro, il lettore potrà così constatare che la chiave se resa [private] non verrà compromessa e di conseguenza le informazioni criptate tramite essa manterranno la propria integrità.

Proprietà di autenticazione

Per dimostrare la proprietà "*authentication*", bisogna enunciarne prima altre due che tornano utili alla comprensione dell'argomento:

- ***Non-injective agreement*** se esistono $e_1(N_1,...,N_k)$ ed $e_2(M_1,...,M_j)$ due eventi e ad ogni esecuzione di $e_1(N_1,...,N_k)$ è avvenuta almeno una volta in precedenza un'esecuzione di $e_2(M_1,...,M_j)$.

La sintassi per esprimere ciò all'interno della grammatica utilizzata è:

query x_1: T_1,..., x_n : T_n; ***event*** ($e_1(N_1, ..., N_k)$) ==> ***event***($e_2(M_1, .., M_j)$). (con $n, j, k \in \mathbb{N}$)

- ***Injective agreement*** se esistono e_1(N_1,...,N_k) ed e_2(M_1,...,M_j) due eventi e ad ogni esecuzione di e_1(N_1,...,N_k) c'è stata in precedenza una distinta esecuzione di e_2(M_1,...,M_j), questo implica che il numero di occorrenze dell'evento e_2(M_1,...,M_j) sia maggiore o uguale rispetto a quelle di e_1(N_1,...,N_k).
  La sintassi per esprimere ciò all'interno della grammatica utilizzata è:

  query x_1: T_1,..., x_n : T_n; ***inj-event*** ($e_1(N_1, ..., N_k)$) ==> ***inj-event***($e_2(M_1, .., M_j)$). (con $n, j, k \in \mathbb{N}$)

Espresse questi due importanti concetti, viene ora elencato com'è stata dimostrata la proprietà.
Innanzitutto, sono stati dichiarati i seguenti eventi nel caso di Server-side Credentials (vedi Codice 5.2) e Client-side Credentials (vedi Codice 5.3).

```
event createPublicKeyCredentialCreationOptions(challenge,bitstring,bitstring).
event createCredential(skey,pkey,challenge,bitstring,bitstring).
event endClientRegistration(pkey,bitstring,bitstring,bitstring).
event endServerRegistration(pkey,bitstring,bitstring).
event createPublicKeyCredentialRequestOptions(challenge,bitstring,bitstring).
event checkCredential(challenge,bitstring,bitstring).
event endClientAuthentication(challenge).
event endServerAuthentication(challenge).
event failedServerAuthentication().
```

Codice 5.2: *Lista di eventi dichiarati nel protocollo formalizzato e con utilizzo di Server-side Credentials*

```
event createPublicKeyCredentialCreationOptions(challenge,bitstring,bitstring).
event createCredential(skey,pkey,challenge,bitstring,bitstring).
event endClientRegistration(pkey,bitstring,bitstring,bitstring).
event endServerRegistration(pkey).
event createPublicKeyCredentialRequestOptions(challenge,bitstring).
event checkCredential(challenge,bitstring).
event endClientAuthentication(challenge).
event endServerAuthentication(challenge).
event failedServerAuthentication().
```

Codice 5.3: *Lista di eventi dichiarati nel protocollo formalizzato e con l'utilizzo di Client-side Credentials*

Ogni evento rappresenta una fase della cerimonia di registrazione o autenticazione, questo intuibile dal nome autoesplicativo.
Per poter dimostrare la proprietà, si è usufruito di quanto asserito in precedenza con "injective agreement", definendo le seguenti query (vedi Codice

5.4, si noti che per brevità sono state riportate le query con eventi espressi in un contesto in cui sono state usate credenziali lato server).

```
query chal:challenge,pubK:pkey,secK:skey,credentialID:bitstring,userID:bitstring
    ; inj-event(endServerRegistration(pubK,credentialID,userID)) ==> inj-event(
    createCredential(secK,pubK,chal,credentialID,userID)).

query chal:challenge,credentialID:bitstring,rpID:bitstring; inj-event(
    endServerAuthentication(chal)) ==> inj-event(checkCredential(chal,
    credentialID,rpID)).
```

Codice 5.4: *Lista di eventi dichiarati nel protocollo formalizzato*

Le seguenti query (vedi Codice 5.4) impongono che la proprietà sia soddisfatta, quindi che queste ritornino *true* dopo che il programma sia stato eseguito, se e solo se:

- l'esecuzione dell'evento rappresentante la fine di una registrazione avvenga sempre e solo nel caso in cui sia preceduto da una sola esecuzione dell'evento rappresentate la creazione della credenziale da parte del client;
- l'esecuzione dell'evento rappresentante la fine di un'autenticazione avvenga sempre e solo nel caso in cui sia preceduto da una sola esecuzione dell'evento rappresentate il controllo da parte del client di una credenziale valida da presentare al server con cui effettuare la cerimonia di autenticazione;

La dimostrazione di questa proprietà applicata al nostro protocollo aiuta a provare la vulnerabilità o meno dell'API ad eventuali replay attack (vedi paragrafo 4.2.2).

Reachability query

Avendo ora gli strumenti per dimostrare le proprietà "secrecy" e "authentication", non rimane che dimostrare che la formalizzazione del protocollo tramite linguaggio sia esente da errori e che la sua esecuzione termini in maniera corretta, per ovviare a ciò è stata definita una *reachability query* la quale fornirà in output valore *false* nel caso in cui l'evento finale sia raggiungibile dall'esecuzione del protocollo trascritto (vedi Codice 5.5).

```
    query chal:challenge; event(endServerAuthentication(chal)).
```

Codice 5.5: *Reachability query per dimostrare che l'esecuzione del protocollo formalizzato arrivi all'evento finale*

Codice

Per facilitare la lettura viene elencato il codice seguendo il percorso che l'informazione compie all'interno dell'API, dando una breve spiegazione nei vari step.
Il flusso del protocollo parte dall'utente che intende registrarsi al sito web compilando un ipotetico form (vedi Codice 5.6), per evitare ridondanze è stato mostrato solo il caso in cui vengono utilizzate credenziali lato client, l'altra tipologia di credenziale invece prevede solamente un ulteriore parametro oltre a quelli passati al processo e mostrato nel codice, cioè la chiave di cifratura di un autenticatore (*keyAuth : key*) .

```
let client(sakAuth : attskey, pakAuth : attpkey, publicKey : pkey, secretKey :
    skey, crID : bitstring) =
  (*INIZIO REGISTRAZIONE*)
  new info : bitstring; (*Compilo form, info contiene tutti i   dati
   sintetizzati*)
  out(c,sencChannel(info,k)); (*Invio i dati al server*)
```

Codice 5.6: *Formalizzazione di parte del client attraverso un dialetto di Applied Pi Calculus ed utilizzando Client-side Credentials*

Una volta compilato, il browser manda al server le informazioni inserite dall'utente all'interno del form, il server ora deve preparare tutto il necessario (*PublicKeyCredentialCreationOptions*) da fornire al client, in modo che questo possa creare delle credenziali (vedi Codice 5.7).

```
let server(rpID : bitstring) =
  (*REGISTRAZIONE*)
  in(c,info : bitstring); (*Ricevo i dati del form compilato  dall'utente*)
  new chal:challenge; (*Creo una challenge per la cerimonia*)
  new userID:bitstring;
  event   createPublicKeyCredentialCreationOptions(chal,userID,rpID);
  out(c,sencChannel((chal,rpID,userID),k)); (*Mando al client
   PublicKeyCredentialCreationOptions*)
```

Codice 5.7: *Formalizzazione di parte del server attraverso un dialetto di Applied Pi Calculus*

Mandato *PublicKeyCredentialCreationOptions* al client, questo comunicherà con l'autenticatore per la creazione di una credenziale (si ricorda che l'autenticatore non è stato rappresentato come entità e per semplicità la credenziale viene creata all'interno del client) che verrà mandata al server una volta prodotta (vedi Codice 5.8 e 5.9).

```
in(c,pubKeyCredCreat : bitstring);  (*Ricevo
PublicKeyCredentialCreationOptions dal server*)
let (chal:challenge,rp:bitstring,userID:bitstring) = sdecChannel(
pubKeyCredCreat,k) in
(*Ricevendo publicKeyCredentialCreationOptions dal server, creo le
credenziali attraverso l'autenticatore*)
event   createCredential(secretKey,publicKey,chal,crID,userID);
(*Create le credenziali, creo PublicKeyCredential e mando   al server per
validazione*)
out(c,sencChannel(signAtt((publicKey,chal,pakAuth,crID,rp),sakAuth),k));
event endClientRegistration(publicKey,crID,rp,userID);
```

Codice 5.8: *Formalizzazione di parte del client attraverso un dialetto di Applied Pi Calculus e con utilizzo di Client-side Credentials*

```
in(c,pubKeyCredCreat : bitstring);  (*Ricevo PublicKeyCredentialCreation dal
  server*)
let (chal:challenge,rp:bitstring,userID:bitstring) = sdecChannel(
pubKeyCredCreat,k) in
(*Creo crID dalla cifratura di Public Key Credential Source*)
let crID = senc(secretKey,userID,rp,keyAuth) in
(*Ricevendo publicKeyCredentialCreationOptions dal server,  creo le
credenziali attraverso l'autenticatore*)
event   createCredential(secretKey,publicKey,chal,crID,userID);
(*Create le credenziali, creo PublicKeyCredential e mando   al server per
validazione*)
out(c,sencChannel(signAtt((publicKey,chal,pakAuth,crID,rp),sakAuth),k));
event endClientRegistration(publicKey,crID,rp,userID);
```

Codice 5.9: *Formalizzazione di parte del client attraverso un dialetto di Applied Pi Calculus e con utilizzo di Server-side Credentials*

Mandata la credenziale al server, questo si occuperà di controllare la sua validità (vedi Codice 5.10).

```
in(c,m : bitstring); (*ricevo m, PublicKeyCredential*)
let message = sdecChannel(m,k) in (*decodifico quanto ricevuto nel canale di
  comunicazione*)
let (pkUser:pkey,challengeReceived:challenge,pubAttkey: attpkey,credentialID
:bitstring,rp:bitstring) = getmessAtt(message) in
if chal = challengeReceived && rp = rpID then
  (let value = checksignAtt(message,pubAttkey) in
  event endServerRegistration(pkUser,credentialID,userID);
```

Codice 5.10: *Formalizzazione di parte del server attraverso un dialetto di Applied Pi Calculus*

Se tutto è andato a buon fine, la registrazione è completata, all'utente ora non rimane che autenticarsi attraverso la credenziale appena validata.
Per brevità è stato tralasciato il passaggio in cui il client compila il form di login o fa sapere al Relying Party di volersi autenticare all'applicativo, quindi quanto mostrato rappresenta il momento in cui il server ha già capito le intenzioni dell'utente e si prepara a fornire ad esso *PublicKeyCredentialRequestOptions* (vedi Codice 5.11 e 5.12).

```
(*INIZIO AUTENTICAZIONE*)
new chal2:challenge; (*Creo una challenge per la  cerimonia*)
event createPublicKeyCredentialRequestOptions(chal2,rpID, credentialID);
out(c,sencChannel((chal2,rpID,credentialID),k)); (*Mando al client
 PublicKeyCredentialRequestOptions*)
```

Codice 5.11: *Formalizzazione di parte del server attraverso un dialetto di Applied Pi Calculus con l'utilizzo di Server-side Credentials*

È bene distinguere il caso in cui si utilizzino credenziali lato server (Codice 5.11) o lato client (Codice 5.12), si noti la differenza tra i due dove in output manca *credentialID*, ID della credenziale associata all'utente, nel caso di Client-side Credentials.

```
(*INIZIO AUTENTICAZIONE*)
new chal2:challenge; (*Creo una challenge per la cerimonia*)
event createPublicKeyCredentialRequestOptions(chal2,rpID);
out(c,sencChannel((chal2,rpID),k)); (*Mando al client
 PublicKeyCredentialCreationOptions*)
```

Codice 5.12: *Formalizzazione di parte del server attraverso un dialetto di Applied Pi Calculus con l'utilizzo di Client-side Credentials*

Una volta fornito *PublicKeyCredentialRequestOptions* al client, questo recupera la credenziale con cui effettuare l'accesso e la manda al server per validazione (vedi Codice 5.13 e 5.14).

```
(*AUTENTICAZIONE*)
in(c,PublicKeyCredOpt : bitstring);
let (chal2:challenge,rp2:bitstring,cr2:bitstring) = sdecChannel(
 PublicKeyCredOpt,k) in
(*Ricevendo publicKeyCredentialRequestOptions dal server, ricavo la
 credenziale con cui accedere*)
let (sec : skey, userid : bitstring, rpid : bitstring) = sdec(cr2, keyAuth)
 in (*Decifro l'ID della credenziale, ottenendo una Public Key Credential
 Source*)
event checkCredential(chal2,cr2,rp2);
(*Ricavata, creo PublicKeyCredential e mando al server per validazione*)
out(c,sencChannel(sign((chal2,cr2,rp2),sec),k));
event endClientAuthentication(chal2).
```

Codice 5.13: *Formalizzazione di parte del client attraverso un dialetto di Applied Pi Calculus con l'utilizzo di Server-side Credentials*

Anche qui bisogna distinguere il caso di credenziale utilizzata, quanto riportato in Codice 5.13 corrisponde all'utilizzo di Server-side Credentials, nel caso di credenziali lato client mancherà in input il valore *cr2* corrispondente all'ID della credenziale da usare, quindi mancherà anche la decifrazione di questo per ottenere una *Public Key Credential Source* ed, inoltre, anche l'evento *checkCredential* non avrà tale valore passato come parametro. Dal processo in output, inoltre, non verrà fornito *cr2*, che si ricorda essere stato

passato dal server, ma una nuova variabile rappresentante sempre l'ID di una credenziale che viene ora recuperato dalla memoria interna dell'autenticatore (vedi Codice 5.14).

```
(*AUTENTICAZIONE*)
in(c,PublicKeyCredOpt : bitstring);
let (chal2:challenge,rp2:bitstring) = sdecChannel(PublicKeyCredOpt,k) in
(*Ricevendo publicKeyCredentialRequestOptions dal server, cerco le
 credenziali con cui accedere*)
event checkCredential(chal2,rp2);
(*Trovate le credenziali, creo PublicKeyCredential e mando al server per
 validazione*)
out(c,sencChannel(sign((chal2,crID,rp2),secretKey),k));
event endClientAuthentication(chal2);
```

Codice 5.14: *Formalizzazione di parte del client attraverso un dialetto di Applied Pi Calculus con l'utilizzo di Client-side Credentials*

Ricevuto *PublicKeyCredential*, il server procederà alla sua validazione (vedi Codice 5.15).

```
in(c,m2 : bitstring); (*ricevo m2, PublicKeyCredential*)
let message2 = sdecChannel(m2,k) in
let value2 = checksign(message2,pkUser) in
let (challengeReceived2:challenge,crID2Received:bitstring,rp2:bitstring) =
 getmess(message2) in
if chal2 = challengeReceived2 && rp2 = rpID then
  event endServerAuthentication(chal2);
else
  event failedServerAuthentication)
```

Codice 5.15: *Formalizzazione di parte del server attraverso un dialetto di Applied Pi Calculus*

Se l'evento *endServerAuthentication* risulta raggiungibile, allora l'autenticazione è terminata correttamente.

5.2.5 Risultati

Una volta formalizzata l'API nell'apposito linguaggio, gettando le basi per la dimostrazione delle proprietà "secrecy" (vedi paragrafo 5.2.4) e "authentication" (vedi paragrafo 5.2.4), simulando così un network attacker intento ad attuare man-in-the-middle e/o replay attack (vedi paragrafo 5.2.1), quindi concentrandosi sull'aspetto della sicurezza di WebAuthn, non rimane che mostrare i risultati delle esecuzioni di quanto espresso fino ad ora.
Come ribadito in precedenza, il protocollo è stato scritto in due versioni, a seconda che siano utilizzate credenziali lato server o client, anche se, come verrà mostrato, al fine della sicurezza davvero poco o nulla si differenzia come risultato finale, influendo più sull'aspetto della privacy che in altri. Per dimostrare anche l'importanza del requisito di implementazione di WebAuthn API in ambienti sicuri dove la comunicazione risulta essere crittografata, sono stati effettuati dei test diversificando l'ambiente, sia nel caso in cui sia

protetto da crittografia che non, simulando l'utilizzo di un protocollo come HTTPS.
L'output del programma nel caso in cui si utilizzino Server-side Credentials ed un canale di trasmissione protetto da crittografia è quanto mostrato in Codice 5.16.

```
Query inj-event(endServerRegistration(pubK,credentialID_1,userID_2)) ==> inj-
    event(createCredential(secK,pubK,chal_2,credentialID_1,userID_2)) is true.

Query inj-event(endServerAuthentication(chal_2)) ==> inj-event(checkCredential(
    chal_2,credentialID_1,rpID_2)) is true.

Query not event(endServerAuthentication(chal_2)) is false.

Query not attacker(pakAuth[]) is true.

Query not attacker(sakAuth[]) is true.

Query not attacker(publicKey[]) is true.

Query not attacker(secretKey[]) is true.

Query not attacker(keyAuth[]) is true.

Query not attacker(crID[]) is false.

Query not attacker(k[]) is true.
```

Codice 5.16: *Risultato dell'esecuzione del programma in cui è stato formalizzato il protocollo, con utilizzo di Server-side Credentials e canale sicuro*

Come si può vedere, le query dimostranti la proprietà di autenticazione ritornano entrambe valore *true*, quindi il protocollo non è vulnerabile a replay attack, inoltre, l'integrità e la segretezza di quasi tutte le informazioni interessate non è stata compromessa ed un attacco man-in-the-middle non avrebbe successo, quasi tutte poichè "crID", identificatore della credenziale, può essere facilmente scoperto con questo tipo di credenziali (vedi paragrafo 4.3.4), venendo di conseguenza modellato come informazione già a conoscenza in partenza dall'attaccante (vedi paragrafo 5.2). Ciò nonostante, il contenuto che rappresenta l'ID della credenziale attraverso cifratura simmetrica, al cui interno si trova anche la chiave privata di una credenziale *secretKey*, non viene compromesso dato che solo l'autenticatore possiede la chiave necessaria utile alla sua decriptazione.
Similmente a quanto mostrato accade utilizzando Client-side Credentials ma con una piccola differenza (vedi Codice 5.17).

```
Query inj-event(endServerRegistration(pubK,credentialID_1,userID_2)) ==> inj-
  event(createCredential(secK,pubK,chal_2,credentialID_1,userID_2)) is true.

Query inj-event(endServerAuthentication(chal_2)) ==> inj-event(checkCredential
  (chal_2,credentialID_1,rpID_2)) is true.

Query not event(endServerAuthentication(chal_2)) is false.

Query not attacker(pakAuth[]) is true.

Query not attacker(sakAuth[]) is true.

Query not attacker(publicKey[]) is true.

Query not attacker(secretKey[]) is true.

Query not attacker(crID[]) is true.

Query not attacker(k[]) is true.
```

Codice 5.17: *Risultato dell'esecuzione del programma in cui è stato formalizzato il protocollo, con utilizzo di Client-side Credentials e canale sicuro*

Rispetto a quanto accaduto in Codice 5.16, l'integrità di *crID* non viene compromessa dato il non passaggio durante la cerimonia di autenticazione di alcun ID identificante una credenziale tramite l'array *allowCredentials* (vedi paragrafo 3.3.2 e 4.3.4).
La reachability query, invece, in tutti i casi fornisce come output "false", quindi l'esecuzione della formalizzazione del protocollo riesce a raggiungere l'evento *endServerAuthentication* che corrisponde, viene ricordato, all'autenticazione di un utente all'interno di un'applicazione web.
Riassumendo, sia con credenziali lato client che server, il protocollo mantiene integri i dati transitanti (tranne l'ID delle credenziali con Server-side credentials), garantendo agli utenti utilizzatori ed ai Relying Party implementatori un servizio robusto e sicuro, in linea con quanto dimostrato in *(Guirat et al., 2018)*[17] ottenendo gli stessi risultati per quanto riguarda la sicurezza del protocollo.
Viceversa, con il non utilizzo di un canale protetto da crittografia, i risultati sono molto differenti rispetto a quanto visto fino ad ora, la sicurezza del protocollo è così minata.
Infatti, sia nel caso di Client che di Server-side Credentials, il risultato ottenuto non è incoraggiante (vedi Codice 5.18 e 5.19).

```
Query inj-event(endServerRegistration(pubK)) ==> inj-event(createCredential(secK
    ,pubK,chal_2,credentialID,userID_2)) is false.

Query inj-event(endServerAuthentication(chal_2)) ==> inj-event(checkCredential(
    chal_2,rpID_2)) is false.

Query not event(endServerAuthentication(chal_2)) is false.

Query not attacker(pakAuth[]) is false.

Query not attacker(sakAuth[]) is true.

Query not attacker(publicKey[]) is false.

Query not attacker(secretKey[]) is true.

Query not attacker(crID[]) is false.

Query not attacker(k[]) is false.
```

Codice 5.18: *Risultato dell'esecuzione del programma in cui è stato formalizzato il protocollo con canale non sicuro e con utilizzo di Client-side Credentials*

```
Query inj-event(endServerRegistration(pubK)) ==> inj-event(createCredential(
    secK,pubK,chal_2,credentialID,userID_2)) is false.

Query inj-event(endServerAuthentication(chal_2)) ==> inj-event(checkCredential
    (chal_2,rpID_2)) is false.

Query not event(endServerAuthentication(chal_2)) is false.

Query not attacker(pakAuth[]) is false.

Query not attacker(sakAuth[]) is true.

Query not attacker(publicKey[]) is false.

Query not attacker(secretKey[]) is true.

Query not attacker(keyAuth[]) is true.

Query not attacker(crID[]) is false.

Query not attacker(k[]) is false.
```

Codice 5.19: *Risultato dell'esecuzione del programma in cui è stato formalizzato il protocollo con canale non sicuro e con utilizzo di Server-side Credentials*

Come si può notare, il non utilizzo di crittografia nel canale di comunicazione comporta un protocollo vulnerabile a man-in-the middle e replay attacks, in linea con quanto detto nel paragrafo 4.2.1 e 4.2.2, rendendo attaccabile la chiave pubblica di una credenziale, quella pubblica di attestazione dell'autenticatore e l'ID della credenziale con cui si ha effettuato una od entrambe le cerimonie. Nonostante l'ID della credenziale sia ora visibile all'attaccante con entrambe le tipologie di credenziale, nel caso di Server-side Credentials,

cioè quindi nel caso in cui l'ID sia la cifratura di una *Public Key Credential Source*, la chiave privata assocciata alla credenziale e più in generale l'intera struttura non viene compromessa, mantenendo quindi un minimo di integrità dei dati transitanti nonostante il canale non sicuro sia stato compromesso.

Capitolo 6

Conclusioni

Nel seguente documento ci si è impegnati a presentare FIDO2 ed in dettaglio WebAuthn API, trattando il suo possibile utilizzo da parte di un Relying Party, parlando delle sue problematiche, analizzando a livello teorico le principali vulnerabilità riscontrate con le relative mitigazioni e supportando questo con un'analisi formale di sicurezza effettuata tramite ProVerif, software che permette la formalizzazione e la verifica della sicurezza di un protocollo.
In dettaglio, nell'analisi teorica effettuata ci si è concentrati in attacchi che potrebbero minare la sicurezza del protocollo e/o la privacy dell'utente e che godessero di una pericolosità non trascurabile, scoprendo come il protocollo sia resistente "by design" a replay attack e phishing e, solo nel caso in cui la chiave pubblica di una credenziale non sia stata compromessa nella registrazione, un attaccante non può rubare in alcun modo l'identità dell'utente (ovviamente tranne nel caso in cui l'autenticatore venga sottratto al proprietario).
Caso diverso è stato osservato con il Cross-site Scripting, introducendo questo attacco si è simulato a livello teorico come potrebbe agire questo al fine di compromettere l'integrità di un sistema che si basa su autenticazione FIDO2, si è così delineata una certa vulnerabilità del protocollo a questo attacco, spetterà quindi agli sviluppatori rendere sicura la propria applicazione che si appoggia a FIDO2.
Con la lettura di questi attacchi il lettore potrà notare come la superficie su cui un attaccante può fare affidamento per lanciare un attacco sia comunque nettamente inferiore rispetto a quella che può avere con una normale autenticazione con password, inoltre questa è ulteriormente riducibile attraverso un'implementazione intelligente di alcune best practices, ciò permette all'API di rendere l'autenticazione senza l'ausilio di password molto più robusta di quanto lo sia utilizzandola, incrementando contemporaneamente l'usabilità che un utente percepisce, un esempio su tutti è l'utilizzo di Client-side Credentials che permette l'eliminazione di gran parte delle vulnerabilità ri-

guardanti la privacy di un utente.
Per dare supporto a quanto detto fino ad ora, si è svolta un'analisi formale sulla sicurezza del protocollo ed è stato dimostrato tramite ProVerif come un network attacker non abbia possibilità di successo nel violare l'integrità del protocollo, dato il requisito di dover utilizzare questo solamente in ambienti protetti da crittografia, quindi HTTPS è una panacea nei threat attuabili da un network attacker.
Tutte queste premesse garantiscono una buona base di partenza da cui lanciare verso il successo questo nuovo standard ma si è visto che, nonostante ciò, c'è una certa riluttanza nell'adottarlo, sia per una mancata totale compatibilità con i browser, ma anche per una non implementazione quando possibile da parte dei soggetti possessori di siti web e, non ultima come importanza, se una persona non dispone di un autenticatore nativo all'interno del dispositivo che utilizza, deve spendere una certa somma di denaro per poter usufruire di questo tipo di autenticazione, rendendo la gente restia a questa tecnologia, essendo che l'obiettivo di registrarsi o autenticarsi ad un servizio web lo può raggiungere gratuitamente attraverso l'utilizzo di sistemi alternativi.

Sitografia e Bibliografia

Sitografia

[1] Ackermann, Y. (2019, Gennaio 15). Introduction to WebAuthn API. *Medium*. Tratto da https://medium.com/webauthnworks/introduction-to-webauthn-api-5fd1fb46c285

[2] Can I Use. (n.d.). Tratto da https://caniuse.com/

[3] CipherBlade. (2019, Giugno 5). The SIM swapping bible: what to do when SIM-swapping happens to you. *Medium*. Tratto da https://medium.com/mycrypto/what-to-do-when-sim-swapping-happens-to-you-1367f296ef4d#47a2

[4] CBOR Object Signing and Encryption (COSE). (2017, Gennaio 11). Tratto da https://www.iana.org/assignments/cose/cose.xhtml

[5] Clickjacking. (n.d.). Tratto da https://blog.intigriti.com/hackademy/clickjacking/

[6] Downgrade attack. (n.d.). Tratto da https://encyclopedia.kaspersky.com/glossary/downgrade-attack/

[7] Herrjemand. (2019, Gennaio 11). Attestation privacy advice creates large scale security risks. *GitHub*. Messaggio postato su https://github.com/w3c/webauthn/issues/1127

[8] Rosenthal, M. (2022, Gennaio 12). Must-know phishing statistics: updated 2022. *Tessian*. Tratto da https://www.tessian.com/blog/phishing-statistics-2020/

[9] Schaad, J., & Cellars, A. (2017, Luglio). CBOR object signing and encryption (COSE). *IETF Datatracker*. Tratto da https://datatracker.ietf.org/doc/html/rfc8152

[10] Swanagan, M. (2021, Novembre 27). How to prevent the top cyber attacks in 2021. *Purplesec*. Tratto da https://purplesec.us/prevent-cyber-attacks/

[11] USB dongle authentication. (n.d.). Tratto da `https://www.dongleauth.info/`

[12] WebAuthn client registration. (n.d.). Tratto da `https://developers.yubico.com/WebAuthn/WebAuthn_Developer_Guide/WebAuthn_Client_Registration.html`

[13] Web authentication (WebAuthn). (2020, Giugno 11). Tratto da `https://www.iana.org/assignments/webauthn/webauthn.xhtml`

[14] Webmaster. (2019, Settembre 12). Attacco "man in the middle": che cos'è e come funziona?. *Apolis*. Tratto da `https://apolis.it/2019/09/man-in-the-middle-come-funziona/`

Bibliografia

[15] Abadi, M., & Fournet, C. (2001). *Mobile values, new names, and secure communication*. In ACM SIGPLAN Notices, Vol. 36. ACM, 104-115. `https://doi.org/10.1145/373243.360213`

[16] Brust, T. (2019). *Security evaluation of multi-factor authentication in comparison with the web authentication API* [Tesi di laurea, Hochschule Wismar]. `https://github.com/timbru31/wings/raw/master/security_evaluation_of_multi-factor_authentication_in_comparison_with_the_web_authentication_api.pdf`.

[17] Guirat,I.B., & Halpin, H. (2018). *Formal verification of the W3C web authentication protocol*. In HoTSoS '18: Proceedings of the 5th Annual Symposium and Bootcamp on Hot Topics in the Science of Security. `https://doi.org/10.1145/3190619.3190640`

[18] Morris, R., & Thompson, K. (1978). *Password security: a case history*. In Communications of the ACM, Volume 22 Issue 11. `https://doi.org/10.1145/359168.359172`

Elenco delle figure

2.1 *Illustrazione di autenticazione tramite FIDO2.* 10
2.2 *Esempio di autenticatore Cross-Platform;* 11
2.3 *Illustrazione di un'autenticazione con l'ausilio di password.[1]* 12
2.4 *Illustrazione di un'autenticazione senza l'ausilio di password.[1]* 12

3.1 *Illustrazione di una cerimonia di registrazione. A=autenticatore, C=client, S=server* 19
3.2 *Esempio di verifica di un utente tramite autenticatore Cross-Platform durante una registrazione.* 22
3.3 *Composizione di attestationObject.[12]* 23
3.4 *Illustrazione di una cerimonia di autenticazione. A=autenticatore, C=client, S=server* 27
3.5 *Esempio di test di presenza durante un'autenticazione.* 29
3.6 *Interfaccia di Office 365 in cui è permesso all'utente di accedere tramite chiave di sicurezza.* 33

4.1 *Illustrazione di man-in-the-middle.[14]* 35
4.2 *Illustrazione di replay attack.[10]* 36
4.3 *Illustrazione di Phishing.[10]* 38
4.4 *Illustrazione artistica di clickjacking.[5]* 39

Codice

3.1 *Esempio di PublicKeyCredentialCreationOptions* 21
3.2 *Esempio di oggetto che rispetta l'interfaccia PublicKeyCredential* 24
3.3 *Esempio di clientDataJSON.* 25
3.4 Esempio di *attestationObject* 25
3.5 *Esempio di publicKeyCredentialRequestOption* 28
3.6 *Esempio di asserzione fornita al server per validazione* 31
5.1 *Query per dimostrare la proprietà secrecy del protocollo* 55
5.2 *Lista di eventi dichiarati nel protocollo formalizzato e con utilizzo di Server-side Credentials* 56
5.3 *Lista di eventi dichiarati nel protocollo formalizzato e con l'utilizzo di Client-side Credentials* 56
5.4 *Lista di eventi dichiarati nel protocollo formalizzato* 57
5.5 *Reachability query per dimostrare che l'esecuzione del protocollo formalizzato arrivi all'evento finale* 57
5.6 *Formalizzazione di parte del client attraverso un dialetto di Applied Pi Calculus ed utilizzando Client-side Credentials* . . 58
5.7 *Formalizzazione di parte del server attraverso un dialetto di Applied Pi Calculus* 58
5.8 *Formalizzazione di parte del client attraverso un dialetto di Applied Pi Calculus e con utilizzo di Client-side Credentials* . 59
5.9 *Formalizzazione di parte del client attraverso un dialetto di Applied Pi Calculus e con utilizzo di Server-side Credentials* . 59
5.10 *Formalizzazione di parte del server attraverso un dialetto di Applied Pi Calculus* 59
5.11 *Formalizzazione di parte del server attraverso un dialetto di Applied Pi Calculus con l'utilizzo di Server-side Credentials* . 60
5.12 *Formalizzazione di parte del server attraverso un dialetto di Applied Pi Calculus con l'utilizzo di Client-side Credentials* . 60
5.13 *Formalizzazione di parte del client attraverso un dialetto di Applied Pi Calculus con l'utilizzo di Server-side Credentials* . 60
5.14 *Formalizzazione di parte del client attraverso un dialetto di Applied Pi Calculus con l'utilizzo di Client-side Credentials* . 61

5.15 *Formalizzazione di parte del server attraverso un dialetto di Applied Pi Calculus* 61
5.16 *Risultato dell'esecuzione del programma in cui è stato formalizzato il protocollo, con utilizzo di Server-side Credentials e canale sicuro* 62
5.17 *Risultato dell'esecuzione del programma in cui è stato formalizzato il protocollo, con utilizzo di Client-side Credentials e canale sicuro* 63
5.18 *Risultato dell'esecuzione del programma in cui è stato formalizzato il protocollo con canale non sicuro e con utilizzo di Client-side Credentials* 64
5.19 *Risultato dell'esecuzione del programma in cui è stato formalizzato il protocollo con canale non sicuro e con utilizzo di Server-side Credentials* 64

Acronimi e abbreviazioni

2FA Two Factor Authentication

AAGUID Authenticator Attestation Globally Unique Identifier

API Application programming interface

CBOR Concise Binary Object Representation

COSE CBOR Object Signing and Encryption

CTAP2 Client to Authenticator Protocol 2

FIDO2 Fast IDentity Online 2

HTTPS Hypertext Transfer Protocol Secure

JS JavaScript

JSON JavaScript Object Notation

OTP One Time Password

PIN Personal Identification Number

RP Relying Party

rpID Relying Party Identifier

SHA-256 Secure Hash Algorithm-256

SMS Short Message Service

SOP Same Origin Policy

U2F Universal Second Factor

W3C World Wide Web Consortium

XSS Cross-site Scripting