Università degli studi di Modena e Reggio Emilia

Dipartimento di Ingegneria "Enzo Ferrari"

Corso di Laurea in Ingegneria Informatica - Sede di Mantova

Sviluppo prototipale di una soluzione survivable per autenticatori hardware FIDO

Relatore:
Luca Ferretti

Candidato:
Giulio Barabino

Correlatore:
Federico Magnanini

Anno Accademico 2021/2022

Indice

Capitolo 1

Introduzione

Il Single Sign-On è un protocollo utilizzato per migliorare e semplificare il processo di autenticazione. Esso alleggerisce i servizi Web che lo utilizzano dall'onere di gestire la moltitudine di credenziali degli utenti, con i meccanismi di protezione necessari che ne conseguono. I protocolli di SSO consentono l'autenticazione passwordless degli utenti: grazie ad essi l'utente è in grado di eseguire operazioni di autenticazione basate su crittografia asimmetrica tramite un token hardware. Questo permette allo stesso tempo di semplificare e irrobustire il processo complessivo di autenticazione. Il limite principale risiede nella centralizzazione del protocollo SSO, che non è in grado di tollerare intrusioni nell'infrastruttura che lo esegue. Infatti, come mostrano attacchi recenti [5] [1], attaccanti che riescono a compromettere l'infrastruttura di SSO sono in grado di impersonare utenti arbitrari. Per mitigare questa problematica sono nati gli studi sulla cosiddetta SSO *survivability*. Tale filone si pone l'obiettivo di sviluppare un approccio distribuito e replicato per il protocollo SSO. Si prevede un numero di attori malevoli di cui è possibile tollerare l'intromissione, numero da cui dipende il livello di sicurezza.

L'esigenza di questa tesi nasce dal fatto che i protocolli esistenti in ambito SSO *survivable* non permettono di modificare il livello di sicurezza una volta che è stato stabilito durante la fase di creazione dell'infrastruttura. Questo impedisce ai vari servizi Web di richiedere alla stessa infrastruttura SSO livelli di sicurezza adeguati alle caratteristiche del servizio offerto. Si è affrontato dunque il problema dell'integrazione del livello di sicurezza all'interno dei protocolli Passwordless e SSO preesistenti. A tal fine, si è reso necessario implementare modifiche al codice di un autenticatore open source,

cui hanno fatto seguito verifiche delle performance di autenticazione e un breve studio di fattibilità per l'implementazione su hardware reale.

La tesi è strutturata nel seguente modo: il capitolo 2 introduce le conoscenze necessarie per proseguire nella lettura dei successivi; il capitolo 3 modella gli attori che partecipano al protocollo di SSO proposto, il funzionamento dei protocolli citati e la soluzione studiata per integrare il livello di sicurezza; il capitolo 4 approfondisce le modifiche apportate al codice degli attori presenti; il capitolo 5 analizza i dati raccolti, valuta le tempistiche e studia la fattibilità; il capitolo 6 conclude con le considerazioni finali ed eventuali sviluppi futuri.

Capitolo 2

Conoscenze di Base

In questa sezione si introducono i concetti necessari per proseguire con la lettura dei capitoli successivi.

2.1 Crittografia asimmetrica

La crittografia asimmetrica è un tipo di crittografia in cui ogni attore possiede una coppia di chiavi: una **pubblica** e una **privata**. Gli usi della crittografia asimmetrica sono due:

- Cifrare le comunicazioni: tramite la chiave pubblica del destinatario è possibile cifrare un messaggio che solo il destinatario può decifrare usando la propria chiave privata.
- Firmare digitalmente: tramite la propria chiave privata un attore può apporre su un messaggio, autenticandolo, la firma, la quale può essere verificata tramite la chiave pubblica del firmatario.

2.2 Single Sign-On

Il *Single Sign-On* è un protocollo ampiamente diffuso utilizzato per autenticare utenti a servizi Web. Il protocollo è eseguito da un utente, un User Agent, un Identity Provider

e un Service Provider. L'User Agent è un software usato dall'utente che permette l'interazione con un contenuto Web. L'Identity Provider è un attore terzo che si occupa di fornire un sistema di autenticazione sicuro, creando e gestendo le credenziali degli utenti, a servizi Web che lo richiedano. L'Identity Provider possiede molteplici Identity Server presso cui gli utenti devono eseguire le operazioni di autenticazione. Il Service Provider è un fornitore di servizio Web che non implementa sistemi di autenticazione propri, ma si avvale di quelli messi a disposizione dall'Identity Provider. Qualora l'utente si autentichi con successo presso l'Identity Server, gli viene fornito un token di autenticazione firmato dell'Identity Server, che deve presentare al Service Provider per poter fruire di quel servizio. Una volta presentato il token al Service Provider, l'utente risulta autenticato a tutti quei servizi messi a disposizione dal Service Provider che fanno uso dello stesso Identity Provider. Da qui l'accezione *single* di *sign-on*.

2.3 Survivability

La centralizzazione dello schema SSO lascia spazio ad attacchi in cui un malintenzionato prenda il controllo dell'Identity Server e utilizzi la chiave privata dello stesso per firmare token di autenticazione forgiati arbitrariamente, così da poter poi impersonare qualunque utente egli voglia. Giungono in aiuto gli schemi cosiddetti *survivable SSO* che possono limitare tali criticità sfruttando più Identity Server. Un singolo Identity Provider gestisce pertanto più Identity Server e l'utente deve autenticarsi presso un sottoinsieme di questi, i quali rilasciano un **token** firmato collettivamente.

La componente *survivable* risiede nel fatto che si tollera un certo numero di Identity Server violati e, di conseguenza, si richiede un token in funzione di questo numero. Con una soglia di tolleranza di server maligni sufficiente si riesce a garantire l'integrità del meccanismo di autenticazione e un overhead dovuto alla reiterazione dei passaggi trascurabile.

2.4 Passwordless

L'autenticazione passwordless è un metodo di autenticazione che permette all'utente di effettuare il login ad un servizio senza la necessità di conoscere una password o più genericamente senza una conoscenza considerata segreta. Tipicamente utilizza una coppia di chiavi crittografiche, una privata e una pubblica: la prima è generata e immagazzinata sul dispositivo dell'utente, mentre la seconda è inviata al server così che esso possa verificare l'autenticità dei messaggi ricevuti. La chiave privata, o segreta, non lascia mai il dispositivo su cui è stata creata e per accedervi è necessaria l'autorizzazione ottenuta tramite **mediazione** da parte dell'utente. Un'azione è detta mediata da un utente qualora sia necessario il suo esplicito consenso, il quale può avvenire, ad esempio, premendo il bottone sul dispositivo fisico.

La registrazione passwordless e, conseguentemente, l'autenticazione sono svolte seguendo un meccanismo *challenge-response*: al pervenire di una richiesta di registrazione il server invia una cosiddetta *challenge*. L'utente che ha iniziato l'operazione ha il compito di apporre, tramite propria chiave privata, una firma crittografica sulla challenge e di fornire in risposta al server la challenge firmata accompagnata dalla chiave pubblica. In questo modo il server verifica l'autenticità della firma tramite la chiave appena ricevuta e in caso di esito positivo immagazzina la chiave pubblica. La fase di autenticazione è svolta in modo analogo, se non che il server è già in possesso della chiave pubblica e non è quindi necessario inviarla.

Ne deriva che non viene scambiato alcun segreto e l'unica interazione richiesta all'utente è quella in fase di firma della challenge. Anche durante la stessa l'utilizzatore non deve inserire codici o password, ma semplicemente mediare l'operazione tramite uno dei metodi sopra elencati. Sfruttando l'autenticazione passwordless è possibile sopperire alle criticità tipiche dei segreti a bassa entropia come le password, quali phishing, brute forcing etc.

2.5 FIDO

FIDO Alliance è un'associazione nata nel 2013 con lo scopo di migliorare i sistemi di autenticazione tramite la diffusione dell'autenticazione passwordless. Sono gli autori di *FIDO*, un set di specifiche che include gli standard **CTAP** e **WebAuthn**. Nel corso degli anni vi è stato un susseguirsi di iterazioni dello standard: dapprima noto come Universal Authentication Factor, divenne poi Universal 2nd Factor per giungere infine alla versione corrente FIDO 2.0.

Il protocollo CTAP definisce le API che un client può utilizzare per comunicare con un autenticatore. Il protocollo WebAuthn definisce invece le API per l'autenticazione a servizi web sfruttando le chiavi crittografiche.

In particolare, questi protocolli definiscono tutto il necessario per programmare un autenticatore e un server come: strutture dati, metodi, requisiti di funzionamento, encoding dei dati etc.

2.5.1 Rilevamento tentativi di clonazione

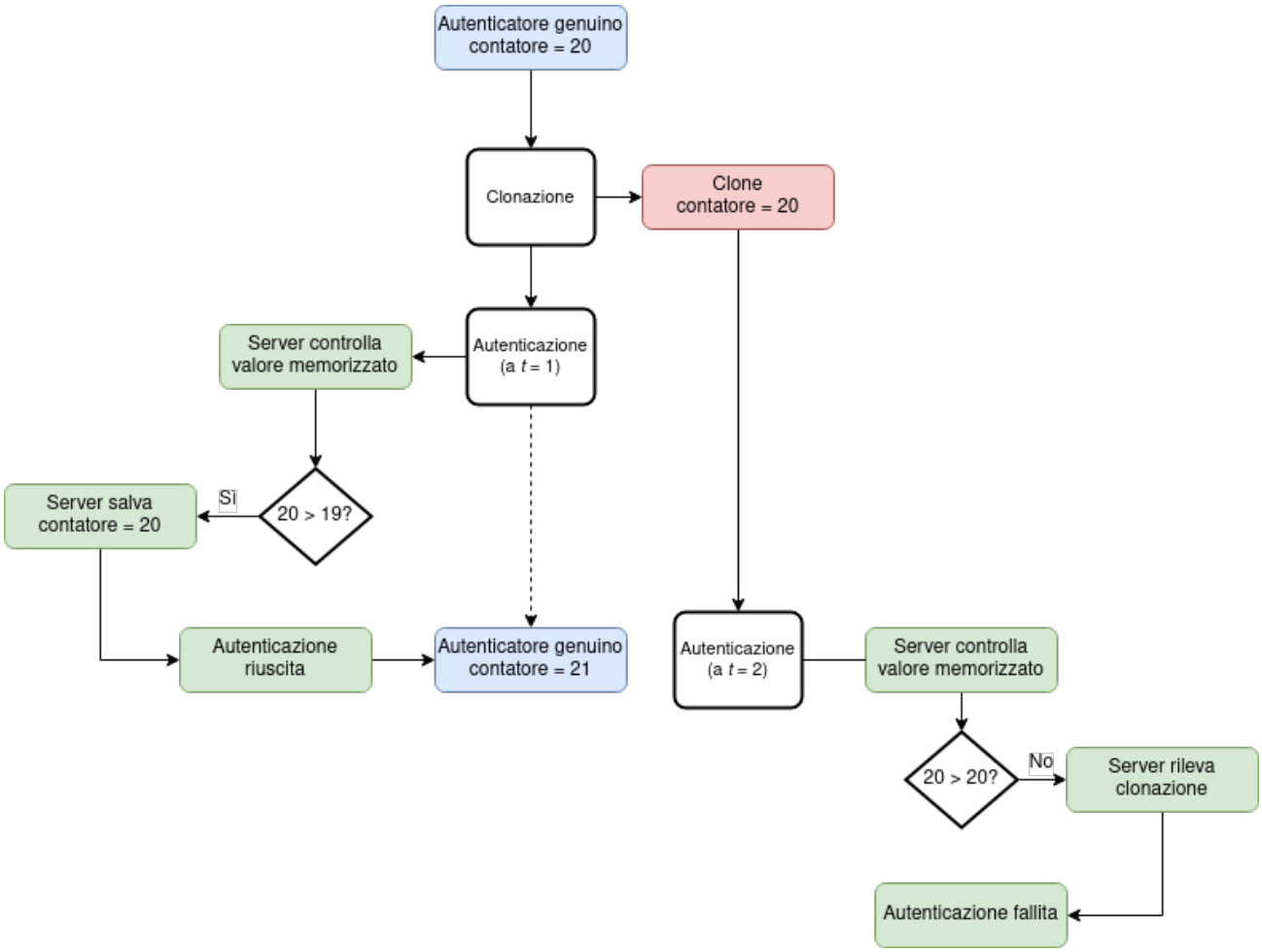
Compito dell'Identity Server è anche quello di rilevare eventuali tentativi di duplicazione dell'autenticatore fisico. Per fare ciò lo standard FIDO prevede un **contatore**, sia esso globale o multiplo, aggiornato dall'autenticatore ad ogni operazione avvenuta con successo. Il contatore prende il nome di *signature counter*. Il server mantiene in memoria l'ultimo valore ricevuto e, all'interazione successiva, controlla che non vi siano discrepanze. Ad esempio, ipotizzando di avere allo stesso tempo:

- Un autenticatore originale con contatore pari a m
- Un autenticatore clone dell'originale con contatore pari a m

Se l'autenticatore originale si autentica presso un servizio, questi aggiorna il proprio contatore a $m + 1$. Il clone, tentando di autenticarsi allo stesso servizio, fornisce un valore del contatore pari a m, dunque minore di quello salvato in memoria dal server al momento dell'ultima interazione con l'autenticatore originale. In questo modo, il server riconosce il clone in quanto tale.

Ne consegue che il rilevamento del tentativo di clonazioni basato sul contatore risulta:

- Inefficace finché il clone non procede ad autenticarsi
- Fallace se il clone procede ad autenticarsi prima dell'originale: quest'ultimo viene di fatto invalidato nonostante sia legittimo



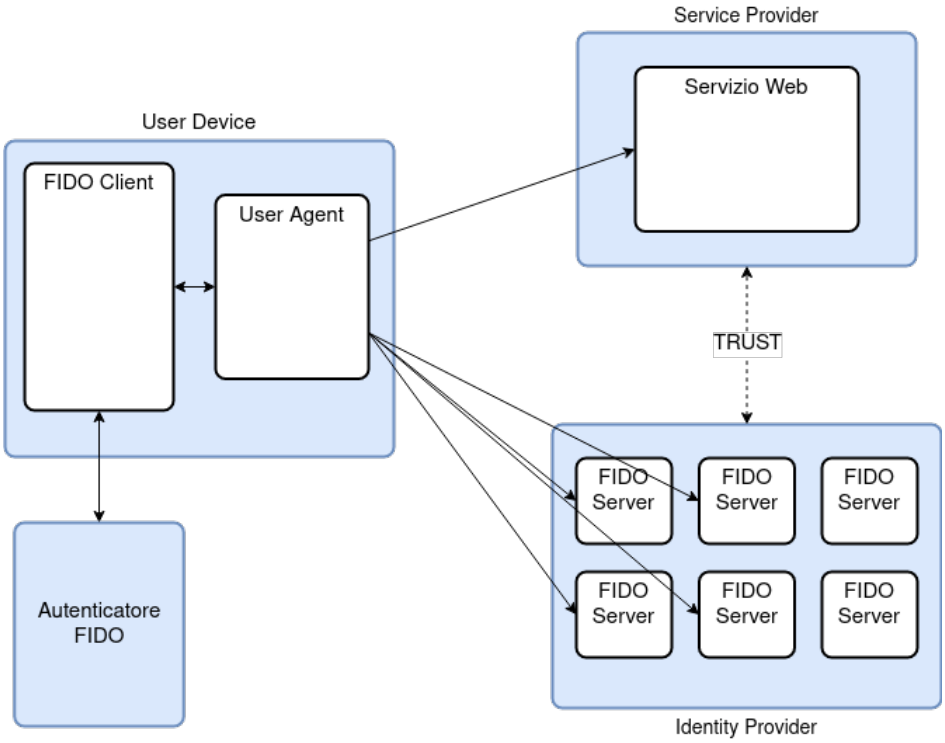
Capitolo 3

Modellazione del sistema

In questo capitolo si descrivono inizialmente gli attori che concorrono alle operazioni di registrazione/autenticazione e in seguito il flusso delle operazioni stesse.

3.1 Attori

Lo schema di seguito riportato rappresenta lo stato attuale dello standard FIDO in accordo all'implementazione descritta successivamente.



Il protocollo include i seguenti attori: l'utente, Service Provider, Identity Provider, un numero n di Identity Server, l'autenticatore hardware e il FIDO Client.

Il **Service Provider** è un fornitore di un generico servizio di cui l'utente è interessato ad usufruire. Può trattarsi di un qualunque servizio di streaming, banking, shopping etc. il quale per l'autenticazione dei propri utenti fa uso di un intermediario a causa di varie ragioni, siano esse economiche o legate alla sicurezza. Il Service Provider definisce il **security level** per indicare il livello di *survivability* desiderato, cioè il numero di Identity Server necessari a completare le operazioni di autenticazione/registrazione. Tale valore è un intero positivo ed è stabilito in funzione della confidenzialità del servizio erogato.

L'**Identity Provider** è un ente terzo che si occupa di fornire a un Service Provider il servizio di autenticazione. A tale scopo si avvale di n Identity Server. Ulteriore compito dell'Identity Provider è quello di fornire il token di autenticazione al client da presentare al Service Provider per fare in modo che l'utente possa accedere al servizio scelto. I vari Identity Server memorizzano le credenziali degli utenti ed una chiave segreta con cui autenticare token di autenticazione.

L'**autenticatore** hardware è un dispositivo che, tramite l'interazione con l'utente, permette l'accesso al servizio web richiesto. L'autenticatore si occupa di memorizzare le credenziali. Ogni credenziale contiene una chiave privata, metadati e un **array associativo** di contatori arr_C. L'arr_C associa ad un determinato *security level* il valore di un *contatore* intero per tenere traccia delle operazioni effettuate con successo. L'autenticatore ricorre alla chiave privata per firmare le challenge che gli sono sottoposte e invia la chiave pubblica al server così che esso possa verificare l'autenticità delle firme. Per la fase di creazione e le successive di autenticazione si richiede all'utente di compiere un'azione: essa può essere la pressione di un pulsante sulla chiavetta stessa, un collegamento NFC oppure ancora l'identificazione tramite impronta digitale. Così facendo l'operazione in corso è autorizzata.

Il FIDO **Client** è un dispositivo che sfrutta un **User Agent** conforme ad implmentare le specifiche FIDO per il dialogo con l'Identity Server e con l'autenticatore, in collaborazione con l'hardware sottostante su cui è installato l'User Agent, tipicamente

un sistema operativo. Il client è quindi interposto tra il FIDO Server e l'autenticatore fisico, agendo da intermediario. Il suo compito è duplice:

- Comunicare con il server al fine di iniziare e successivamente terminare le operazioni di autenticazione e di creazione delle credenziali
- Comunicare con l'autenticatore allo scopo di creare le chiavi crittografiche e firmare le challenge ricevute dal server

La comunicazione è bidirezionale e segue i protocolli definiti dagli standard: CTAP per l'interazione con l'autenticatore e WebAuthn per la comunicazione con il FIDO Server. Nel caso particolare dell'estensione *survivable* il client si occupa di replicare le operazioni su n FIDO Server distinti, computando l'hash delle challenge ricevute e fornendolo all'autenticatore come un digest unico su cui apportare la firma.

3.2 Flusso operativo

Il flusso operativo si compone di due operazioni distinte: una fase di creazione delle credenziali e una fase di autenticazione dell'utente. Tali operazioni sono svolte, rispettivamente, durante la registrazione al servizio del Service Provider e a tutte le autenticazioni successive.

3.2.1 Fase di registrazione

Alla fase di registrazione partecipano: l'utente, il FIDO Client, lo User Agent, n Identity Server, l'autenticatore.

La fase di registrazione origina a partire dalla richiesta dell'utente, utilizzando un User Agent, di registrarsi ad un servizio, offerto da un Service Provider, che supporti l'autenticazione passworldess, tramite degli Identity Provider. Il Service Provider fornisce all'utente il livello di sicurezza n necessario per completare l'operazione. La fase di creazione dovrà essere replicata dal client su tutti gli n Identity Server presenti. Il processo messo in atto è il seguente:

1. Ogni Identity Server crea il proprio stato interno e la challenge
2. Ogni Identity Server invia al Client una serie di requisiti, secondo cui deve essere svolta la cerimonia di registrazione, e la challenge generata
3. Il Client salva tutte le challenge ricevute in un vettore e computa l'hash dello stesso
4. Il Client effettua una chiamata al metodo opportuno dell'autenticatore, fornendo i requisiti di creazione richiesti dall'Identity Server e il digest computato come challenge
5. L'autenticatore procede a generare la coppia di chiavi crittografiche seguendo le imposizioni del server; invia al client la challenge firmata accompagnata dalla chiave pubblica e il contatore specifico dell'arr_C inizializzato a uno
6. Il Client invia ad ogni Identity Server il vettore con le challenge, l'hash dello stesso, la firma, la chiave pubblica e il signature counter ricevuto
7. Ogni Identity Server controlla che la challenge da lui generata sia presente all'interno del vettore e controlla, computando lui stesso l'hash del vettore, l'integrità di quanto ricevuto. Infine, verifica tramite la chiave pubblica fornitagli l'autenticità della firma
8. Qualora il processo sia andato a buon fine, gli Identity Server salvano le informazioni ricevute (contatore, chiave pubblica, identificatore del client) al proprio interno

3.2.2 Fase di autenticazione

La fase di autenticazione ricalca i passaggi di quella di registrazione con la differenza che gli Identity Server sono già in possesso della chiave pubblica mediante cui verificare l'autenticità della firma apportata alle challenge.

In questa fase è comunicato un security level pari a $|Q|$ da parte del Service Provider, cioè la cardinalità del sottoinsieme di Identity Server con cardinalità $Q : |Q| \leq n$ presso cui è necessario autenticarsi. Questo valore prende in considerazione la tollerabilità alle

intrusioni che ha il Service Provider. In particolare: $|Q| \in [(2k+1), (3k+1)]$, dove k rappresenta il numero di Identity Server di cui si può tollerare la compromissione.

1. Ogni Identity Server crea il proprio stato interno e la challenge
2. Ogni Identity Server invia al Client una serie di requisiti, secondo cui deve essere svolta la cerimonia di autenticazione, e la challenge generata
3. Il Client salva tutte le challenge ricevute in un vettore e computa l'hash dello stesso
4. Il Client effettua una chiamata al metodo opportuno dell'autenticatore, fornendo i requisiti di autenticazione richiesti dall'Identity Server e il digest computato come challenge
5. L'autenticatore incrementa il contatore dell'arr_C specifico; invia poi al Client la challenge firmata e il contatore aggiornato
6. Il Client invia ad ogni Identity Server il vettore con le challenge, l'hash dello stesso, la firma ricevuta e il signature counter
7. Ogni Identity Server controlla che la challenge da lui generata sia presente all'interno del vettore e controlla, computando lui stesso l'hash del vettore, l'integrità di quanto ricevuto. Infine, verifica tramite la chiave pubblica memorizzata precedentemente l'autenticità della firma
8. Ogni Identiy Server controlla che il *signature counter* ricevuto sia maggiore di quello memorizzato in precedenza e in tal caso aggiorna quest'ultimo con il valore appena ricevuto
9. Se i passaggi precedenti sono avvenuti con successo rilasciano al Client il token di autenticazione tramite cui completare l'autenticazione presso il Service Provider

Capitolo 4

Dettagli

In questo capitolo si trattano i dettagli implementativi relativi alle modifiche operate a:

- Una variante della libreria FIDO2 realizzata da Yubico modificata in una tesi precedente per accogliere il meccanismo survivable [9]
- Il codice sorgente dell'autenticatore Solokeys [7] per integrare il *security level*

Nonostante la libreria Yubico fosse già stata modificata in precedenza per adottare la struttura survivable, è stato comunque necessario operare cambiamenti. La libreria FIDO2 si occupa di simulare l'interazione tra un Client FIDO2 e un Server FIDO2 per emulare la registrazione e la successiva autenticazione. Grazie alla modifica apportata precedentemente è possibile simulare un numero arbitrario di Server con cui stabilire la comunicazione e svolgere tali operazioni. Tutto il funzionamento descritto nel capitolo precedente è correttamente gestito, ad esclusione dell'invio del security level.

Lato autenticatore invece si è reso necessario implementare diverse funzionalità: dal parsing del security level nel messaggio inviato dall'User Agent all'autenticatore fino ad arrivare a un contatore globale vero e proprio. Infatti, da standard FIDO2 il signature counter utilizzato per il controllo della clonazione dell'autenticatore può essere anche globale e non specifico per credenziale [4]. Ciò è dovuto alla natura *constraint*, cioè con limitazioni di memoria importanti, degli autenticatori hardware.

4.1 Modifica della libreria FIDO2

La libreria FIDO2 presenta il file `client_multichallenge.py` in cui viene definita la classe `Fido2ClientMultichallenge`, figlia della classe `Fido2Client` nel relativo `client.py`, che permette l'autenticazione WebAuthn simulando un Client FIDO. Presenta due metodi, `make_credential` per realizzare l'operazione di creazione delle credenziali e `get_assertion` per compiere l'operazione di autenticazione.

Rispetto alla condizione di partenza è stato aggiunto il controllo del contatore lato server. Per fare ciò è stato definito all'interno del costruttore di `Fido2ServerMultichallenge` un dizionario tramite cui tenere traccia dei contatori:

```
self.counter_register = {}
```

I metodi interessati della classe sono sostanzialmente due:

- `register_complete`
- `authenticate_complete`

Entrambe le funzioni si occupano di controllare la correttezza dei dati ricevuti dal Client dopo che ha interpellato l'autenticatore. Entrambi i metodi vengono chiamati nel file `client_multichallenge.py` e ad entrambi è stato aggiunto il passaggio del parametro `security_level`.

Nel primo metodo, `register_complete`, viene salvato il contatore ricevuto all'interno del dizionario. Per fare ciò viene creato un secondo dizionario interno al primo, utilizzando il *CredentialID* ricevuto in dall'autenticatore. In questo secondo dizionario sarà presente la coppia `security_level:contatore`.

```
self.counter_register[credential_id]
    =
      {security_level: attestation_object.auth_data.counter}
```

Nel secondo metodo invece vengono effettuati i controlli sulla base di quanto salvato dal server nel proprio dizionario e quanto ricevuto dall'autenticatore. In caso di contatore ricevuto minore o uguale a quello immagazzinato viene sollevato un errore che porta alla terminazione del programma.

```
if auth_data.counter <= self.counter_register[credential_id][security_level]:
    raise ValueError("Counter did not increase.")
```

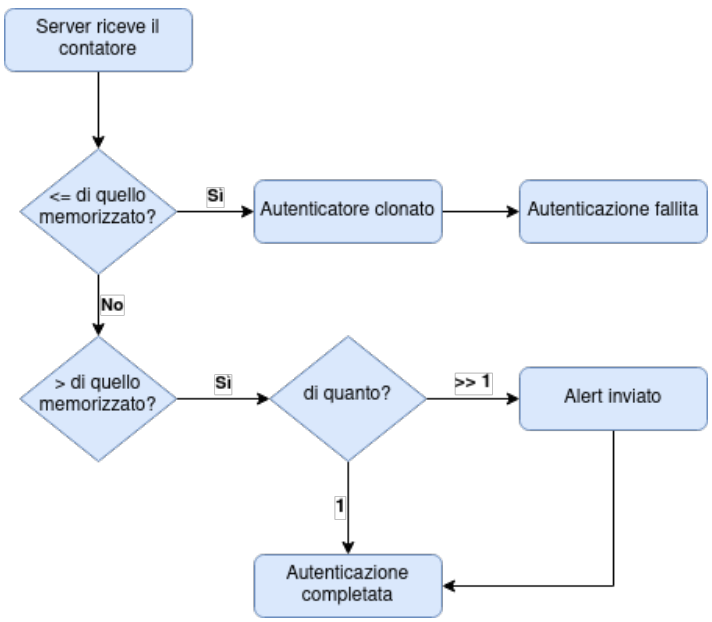
Questo perchè si suppone che vi sia stato un tentativo di clonazione.

Qualora, invece, il contatore fosse maggiore di quello immagazzinato +1 allora verrà solamente notificato il fornitore del server tramite un messaggio in console di log.

```
elif auth_data.counter > self.counter_register[credential_id][security_level] + 1:
    print("Counter bigger than expected")
```

In quest'ultimo caso e nel caso in cui il contatore venga incrementato correttamente viene aggiornato il valore corrispondente nel dizionario del server. Se nessuna di queste condizioni è verificata vuol dire che non è presente nel dizionario una voce con quel dato security level: in questo caso viene semplicemente aggiunto il security level al dizionario interno con il relativo contatore.

```
self.counter_register[credential_id][security_level] = auth_data.counter
```



4.2 Modifica del codice Solo

Il codice dell'autenticatore Solokeys, come detto in precedenza, presentava un solo contatore globale per tenere traccia di tutte le operazioni di creazione/autenticazione svoltesi con successo. Il primo passo è stato, quindi, quello di implementare una struttura dati per la memorizzazione di un contatore per credenziale. Tale struttura dati, definita `signCounter` è una tipo di dato composto da due valori.

- Una istanza `id` della *struct* `CredentialId`
- Un intero senza segno di 32 bit definito come `signCount`

La struttura dati `CredentialId` definisce come vengono memorizzate le credenziali all'interno dell'autenticatore seguendo lo standard FIDO [3]. In particolare, invece che avere una sequenza di 16 bytes come da standard, presenta valori come

`tag, nonce, padding, metadata, rpIdHash`

La definizione della struttura `signCounter` è effettuata all'interno del file `ctap.h` e sempre nello stesso file è inizializzato anche `signCounterArray`, cioè un array di strutture dati `signCounter`. Tramite questo array è possibile memorizzare un contatore per credenziale.

L'array viene popolato nel momento in cui viene chiamata la funzione `ctap_make_credential` utilizzata dall'autenticatore per compiere le operazioni di creazione delle credenziali. In particolare:

- tramite un intero senza segno `globalCounter` viene tenuta traccia dell'ultima posizione dell'array occupata
- viene istanziata una struttura dati `signCounter` con `id` pari a quello calcolato e `signCount` = 1
- viene aggiunta la struttura dati `signCounter` all'array `signCounterArray` alla posizione `globalCounter`

Nel momento in cui viene chiamata la funzione `ctap_get_assertion` viene cercata iterativamente la struttura dati il cui `id` corrisponde a quello dell'operazione corrente e viene incrementato il contatore di tale struct.

```
for (uint l = 0; l <= globalCounter; l++) {
    if (count_cmp_func(&cred->credential.id, &signCounter1[l])) {
        count = update_sign_counter(&signCounter1[l], GA.securityLevel);
    }
}
```

Per verificare che l'`id` corrisponda a quello generato in fase di `get_assertion` è stato necessario scrivere una funzione di comparazione: il C, infatti, non supporta nativamente la comparazione di due struct. A tal scopo è stata scritta la funzione `count_cmp_func` che compara attributo dopo attributo tutti quelli presenti all'interno della struttura per verificarne l'uguaglianza. Il contatore, invece, viene aggiornato tramite la funzione `update_sign_counter` che semplicemente prende il valore `signCount` passato in chiamata e restituisce $signCount + 1$.

Il passo successivo è stato quello di modificare la struttura dati `signCounter` per fare in modo che accogliesse un contatore per *security level*. Per fare ciò l'attributo `signCount` è stato cambiato da intero senza segno di 32 bit ad array di interi senza segno di 32 bit. Analogamente al processo di incremento seguito prima, il contatore corrispondente al security level ricevuto verrà aggiornato nel seguente modo:

```
update_sign_counter(signCounter.signCount[n]])
```

Così facendo vengono mantenuti e incrementati n signature counter differenti per ogni credenziale.

L'ultimo passaggio è stato quello di ricezione del *security level*. Lo standard CTAP2 [2] definisce i codici dei comandi a cui deve essere associato il lancio di alcune funzioni. Ogni comando è strutturato con il proprio codice di comando e i codici per i propri parametri. I codici per per i comandi di interesse sono i seguenti:

```
0x01 authenticatorMakeCredential
0x02 authenticatorGetAssertion
```

L'autenticatore Solo sfrutta il polling per controllare l'arrivo di messaggi da parte del Client. Al giungere di uno di questi viene controllato il codice del comando presente nei primi byte e viene invocata la funzione indicata dal codice e di cui è stato mostrato un esempio sopra. Tale funzione a sua volta effettuerà il parsing, cioè l'analisi del contenuto, per ottenere i parametri della funzione invocata. Oltre ai codici per i parametri già esistenti dei comandi `authenticatorMakeCredential` e `authenticatorGetAssertion` è stato necessario aggiungere nel file `ctap.h`:

```
0x0A MC_securityLevel
0x08 GA_securityLevel
```

Per definire i codici con cui codificare i livelli di sicurezza da usare, rispettivamente, in fase di creazione credenziali (**M**ake**C**redential) e autenticazione (**G**et**A**ssertion).

Solo utilizza delle funzioni definite nel file `ctap_parse.c` per fare il parsing del flusso di dati CBOR ricevuto dal Client. In particolare sono presenti due funzioni:

- `ctap_parse_make_credential`
- `ctap_parse_get_assertion`

che si occupano di controllare il flusso di dati per ottenere i parametri necessari alle funzioni `ctap_make_credential` e `ctap_parse_credential` per poter compiere le operazioni di creazione/autenticazione. In questo caso viene aggiunto allo `switch` l'identificazione dei codici definiti sopra per il parametro *security level*.

Di seguito il funzionamento semplificato all'interno della funzione `ctap_parse_make_credential`:

```
switch(cmd)
{
    ...
    case MC_securityLevel:
        cbor_value_get_int(MC->securityLevel)
    ...
}
```

La struttura dati `MC` rappresenta il risultato del parsing di tutti gli attributi nel flusso di dati ricevuto e viene restituita dalla funzione `ctap_parse_make_credential` al chiamante, in questo caso la funzione `ctap_make_credential`. In questo modo all'interno della funzione `ctap_make_credential`, dove si realizza l'inizializzazione della struttura dati `signCounter` e, conseguentemente del contatore `signCount`, è possibile utilizzare il valore del *security level* ricevuto dal Client e incrementare di conseguenza il contatore corrispondente.

Analogo quanto avviene all'interno della funzione `ctap_parse_get_assertion` con la differenza che la struttura restituita alla funzione `ctap_parse_assertion`, cioè dove si realizza l'incremento del contatore, sarà chiamata `GA`.

4.3 Testing

I test delle modifiche al codice Solo e alla libreria FIDO sono stati effettuati in locale. Per la parte di WebAuthn è stato utilizzato il file `client_multichallenge.py` che simula, utilizzando le classi `Fido2ClientMultichallenge` e `Fido2ServerMultichallenge`, la fase di creazione delle credenziali e poi l'autenticazione di queste con un numero arbitrario di FIDO Server operanti localmente. L'integrazione fatta al codice già presente è stata di definire i livello di sicurezza da utilizzare durante le due operazioni.

Per quanto riguarda l'autenticatore, invece, è stato utilizzato un emulatore scritto da Solo che permette la simulazione di un autenticatore virtuale comunicante tramite protocollo UDP. Questo ha evitato di dover compiere le operazioni tipiche dei dispositivi embedded di *flash* del software sull'autenticatore fisico e ha snellito sensibilmente il processo di test.

Per fare in modo che l'emulatore UDP di Solo comunicasse con il Client FIDO2 della relativa libreria è stato necessario apportare delle modifiche alla classe `CtapHidConnection` che si occupa di definire, per i principali sistemi operativi, i metodi grazie ai quali è possibile interagire con gli *Human Interface Device*. Per HID si intendono genericamente quei dispositivi elettronici che permettono l'interazione, sia essa in input o in output, con un essere umano. Gli autenticatori hardware fanno quindi parte degli HID. In particolare è stato aggiunto il file `udp_backend.py` in cui viene definita

la classe `UdpCtapHidConnection` facendo l'override della classe `CtapHidConnection`. Nello specifico:

- Viene definito un attributo della classe, chiamato `sock`, che rappresenta il *socket* tramite cui avviene la comunicazione UDP

  ```
  self.sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
  ```

  L'elemento fondante del protocollo UDP sono, infatti, i SOCKET, combinazioni di indirizzi IP e porte necessari per la comunicazione.

- Vengono ridefiniti i metodi di scrittura e lettura dei pacchetti utilizzando i metodi della libreria `socket` per l'invio e la ricezione tramite UDP

  ```
  def write_packet(self, data):
      self.sock.sendto(data, self.remote)/CLionProjects/


  def read_packet(self):
      data, host = self.sock.recvfrom(self.descriptor.report_size_out)
  return data
  ```

Per realizzare questa modifica è stato integrato il codice già scritto da Solo per il proprio strumento di interazione a riga di comando con l'autenticatore [8].

Capitolo 5

Prestazioni

In questo capitolo si mostra l'esito in termini di performance delle modifiche apportate. Le prestazioni sono state misurate tramite uno script ereditato dal lavoro precedente sulla libreria FIDO opportunamente modificato per lo scopo.

5.1 Raccolta dei dati

Per prendere le misurazioni relative alle performance è stato utilizzato un calcolatore con sistema operativo Fedora 36, dotato di CPU Intel i5-8250U e un autenticatore CTAP2 emulato virtualmente tramite il backend UDP descritto nel capitolo precedente 4.3. Grazie all'emulatore è stato possibile inibire la mediazione dell'utente richiesta, in fase sia di creazione delle credenziali che di autenticazione. In tal modo, il dato ottenuto risulta deterministico.

I tempi di esecuzione sono stati presi tramite l'ausilio della libreria Python `timeit`, che include metodi per la misura del tempo di esecuzione di funzioni. Tali metodi accettano come parametri la funzione da misurare e il numero di volte che deve essere eseguita, restituendone il tempo di esecuzione. I metodi misurati sono quelli descritti nei capitoli precedenti:

Per la fase di creazione delle credenziali

- `register_begin()`

- `make_credential()`
- `register_complete()`

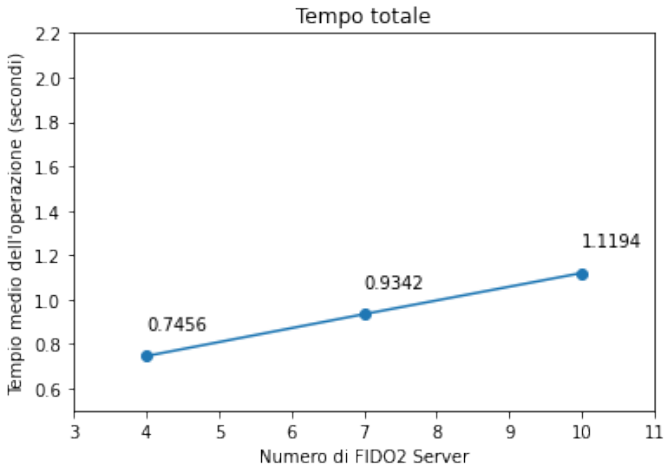
E per la fase di autenticazione

- `authenticate_begin()`
- `get_assertion()`
- `authenticate_complete()`

Per ogni livello di sicurezza $|Q|$ stabilito, il numero n di server presenti e il numero k di server malevoli con $k \in [1, n]$, i test sono stati ripetuti con un livello di sicurezza $|Q| \in [2k + 1, 3k + 1]$. Per diminuire la variabilità dei dati i tempi del metodo `get_assertion()` sono stati presi sulla base della media di dieci iterazioni. La fase di creazione è stata, invece, ripetuta solamente una volta.

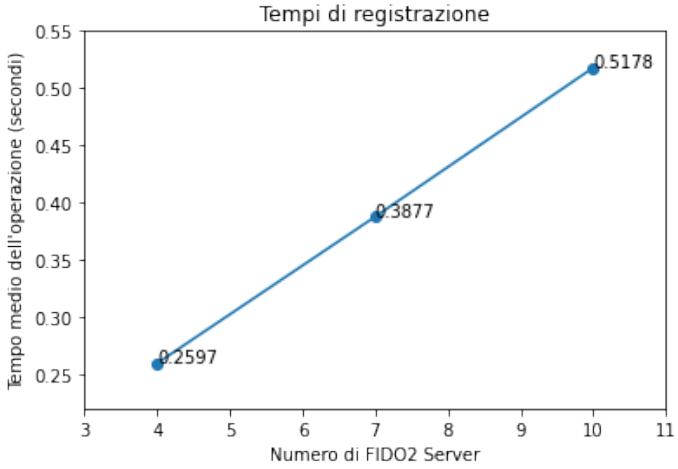
5.2 Valutazione dei dati

I dati nel grafico sono relativi alle fasi sia di creazione che di autenticazione, cioè al tempo totale misurato.

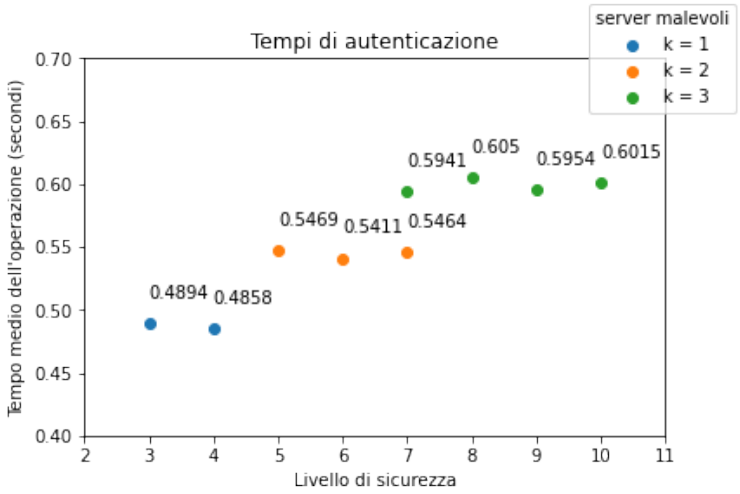


Come si può notare dal grafico all'aumentare del numero di FIDO Server l'operazione totale subisce un incremento di circa 20 *ms*.

I tempi per la fase di registrazione riportano un risultato simile: un incremento di circa 13 *ms* ogni tre FIDO Server aggiuntivi.



Nella figura successiva sono rappresentati i tempi misurati per la sola fase di autenticazione. Per ogni valore di k sono stati utilizzati i valori di security level nel range $[2k + 1, 3k + 1]$. Come si può notare la variazione del livello di sicurezza, dato lo stesso numero di server malevoli tollerabili, non influenza sensibilmente i tempi dell'operazione.



Occorre precisare che le misurazioni effettuate sono assolutamente ottimistiche per quanto riguarda la latenza sia con i FIDO Server, operanti localmente, sia con l'autenticatore, emulato virtualmente. Inoltre, non tengono conto dell'interazione umana richiesta e dell'utilizzo di un Web Browser con le penalizzazioni che ne conseguono.

Si può quindi concludere dai dati raccolti che l'incremento di tempo richiesto dall'operazione con una combinazione di $|Q|$ e n è di pochi ms in più a fronte di una *survivability* migliore. La parte più onerosa in termini di tempo è quella legata alla registrazione, la quale richiede il doppio del tempo al raddoppiare dei FIDO Server con cui dialogare. La fase di autenticazione, invece, si distanzia di soli $12ms$ tra il caso limite inferiore e quello superiore considerato, con un incremento pari circa al 25%.

5.3 Fattibilità

Come specificato nella sezione relativa ai dettagli implementativi 4.3, è stato utilizzato un emulatore dell'autenticatore hardware sviluppato da Solokeys sia per l'implementazione che per la fase di testing. Il codice dell'autenticatore originale è basato sul microcontrollore STM32L432, che offre 256KB di memoria secondaria. L'implementazione suddivide la memoria secondaria nel seguente modo:

- 14 KB per il bootloader
- 226 KB per il codice da eseguire
- 16 KB disponibili per immagazzinare i dati, di cui solo 2048 B adibite alla memorizzazione del contatore

Poiché la memoria disponibile è così ridotta gli sviluppatori hanno previsto il cosidetto *wrapping delle chiavi*: piuttosto che generare una coppia di chiavi crittografiche ad ogni richiesta di registrazione, viene generata una singola chiave crittografica **M** detta *master key* e ad ogni richiesta di registrazione viene computato l'hash di *HMAC(R, M)* dove **R** è un numero generato al momento. Il risultato dell'hash è una chiave crittografica, da cui verrà derivata la corrispondente chiave pubblica. La prima viene usata per firmare e la seconda viene inviata al server insieme ad **R**. Nessuna informazione viene salvata dall'autenticatore: l'unico dato necessario all'autenticatore in fase di autenticazione è **R**, tramite il quale ripeterà il procedimento computando l'*hash* per generare le chiavi.

Dato che l'implementazione proposta si avvale di una struttura dati `signCounter` che presenta:

- Una struttura dati CredentialID di dimensioni pari a 88 bytes
- Un array di contatori interi senza segno di 16 bit l'uno, i quali garantiscono 1 autenticazione al giorno per 179 anni ($(2^{16}) \div 365$), per 3 livelli di sicurezza

Avremo quindi una occupazione di memoria del `signCounter` di circa 94 bytes. Per rispettare i vincoli di memoria dell'autenticatore, nella soluzione proposta, come nel firmware dell'autenticatore originale, dedichiamo 2048 bytes alla memorizzazione dei contatori. In questo modo è possibile immagazzinare circa **21** credenziali differenti. Il risultato ottenuto è in linea con i limiti stabiliti [10] da Yubico per le proprie *resident keys*, una tipologia particolare di credenziali che necessitano di essere salvate in memoria.

Occorre considerare che l'aggiunta di un contatore limita le possibilità dell'hardware, di fatto riducendo il rapporto costo microcontrollore/numero di credenziali. Si può stimare un costo a contatore (21 credenziali * 3 livelli di sicurezza), con un prezzo dell'STM32L432 per grossi stock di \$4.59673, di circa $0,07$ cents, sicuramente non trascurabile per grandi numeri.

La potenzialità di utilizzare la chiavetta per autenticarsi a virtualmente un numero illimitato di servizi Web viene a mancare, ma ciò è compensato da livelli di sicurezza del tutto sufficienti a garantire un incremento di sicurezza notevole (si parla di una resistenza fino a due server compromessi) e un utilizzo giornaliero costante e duraturo. Considerando anche la limitatezza di servizi Web che offrono l'autenticazione passwordless, si può concludere che il limite di ventuno credenziali non è affatto stringente.

Capitolo 6

Conlusioni

Con questa tesi ci si è posto l'obbiettivo di apportare al codice dell'autenticatore Solo e alla libreria FIDO sviluppata da Yubico le opportune modifiche per garantire la *survivability*. La tesi segue la direzione di lavori preesistenti in ambito di autenticazione survivable [6] e del lavoro già svolto in un'altra tesi precedente sulla libreria FIDO. Proprio in quest'ultima tesi si descrivono risultati conformi a quanto ottenuto in questa sede, ovvero prestazioni adatte a scenari reali di autenticazione distribuita. Ciò è segno che l'aggiunta del livello di sicurezza non costituisce elemento di riduzione delle performance.

La soluzione è stata provata solamente in configurazione locale, motivo per il quale sarebbero necessarie ulteriori analisi prevedendo un autenticatore hardware reale e l'utilizzo di un Browser tramite cui simulare l'interazione dell'utente con un servizio Web. Inoltre, sarebbe interessante studiare la fattibilità per altre tipologie di microcontrollori e il rapporto costi/benefici che deriverebbe dall'adozione rispetto a quello preso in esame.

Bibliografia

[1] National Security Agency. Detecting abuse of authentication mechanisms. technical report pp-20-1485. Dec. 2020.

[2] FIDO Alliance. CTAP2 Commands, 27 February 2018.

[3] FIDO Alliance. Credential Id definition, 8 April 2021.

[4] FIDO Alliance. Signature counter considerations, 8 April 2021.

[5] Sean Koessel Steven Adair Thomas Lancaster Volexity Threat Research Damien Cash, Matthew Meltzer. Dark Halo Leverages SolarWinds Compromise to Breach Organizations. 14 December 2022.

[6] Federico Magnanini, Luca Ferretti, and Michele Colajanni. Flexible and Survivable Single Sign-On. 2022.

[7] Solokeys. Solokeys/solo1: Solo 1 firmware in C.

[8] Solokeys. Solo1-cli UDP Backend, 8 March 2022.

[9] Yubico. Yubico's FIDO2 Python Implementation.

[10] Yubico. Yubico Discoverable Keys, 2022.