Università degli Studi di Milano
Facoltà di Scienze e Tecnologie
Corso di Laurea in Sicurezza dei Sistemi e delle Reti Informatiche

DAL TRASPORTO SICURO ALL'ACCESSO CONTROLLATO: UN APPROCCIO COMPLETO ALLA SICUREZZA DEI SISTEMI WEB MODERNI

Relatore: Professoressa Chiara Braghin

Correlatore: Maurizio Cavalieri

Tesi di:
Lorenzo Fallani
Matricola: 984494

Anno Accademico 2023-2024

Indice

Introduzione

Il tema della sicurezza dei sistemi web moderni è diventato di fondamentale importanza con l'evoluzione e la diffusione di internet. Con il tempo, l'aumento delle minacce informatiche e la necessità di proteggere i dati hanno reso indispensabile l'adozione di protocolli più sicuri e complessi.

Questa tesi analizza l'evoluzione dei protocolli di sicurezza, da quelli per garantire il trasporto sicuro, fino a quelli per l'accesso controllato, esaminando le diverse metodologie di autenticazione e autorizzazione adottate nei sistemi web moderni. Inoltre, si esploreranno le capacità di sicurezza integrate nei server web e le tecniche utilizzate per instaurare canali di comunicazione sicuri tra client e server. L'obiettivo principale è fornire una panoramica completa e approfondita delle soluzioni esistenti e delle best practice per garantire la sicurezza dei sistemi web, offrendo spunti e soluzioni pratiche per affrontare le sfide odierne nel campo della cybersecurity.

Il lavoro è il frutto di un approfondito studio teorico unito a una serie di implementazioni pratiche, volte a dimostrare l'efficacia delle soluzioni proposte. Durante lo sviluppo di questa tesi, sono stati esaminati vari strumenti e tecniche, con particolare attenzione all'utilizzo del tool OpenSSL per la gestione della crittografia nelle comunicazioni. È stato inoltre analizzato il ruolo cruciale dei certificati digitali e delle infrastrutture a chiave pubblica (PKI) nel garantire l'integrità e l'autenticità delle comunicazioni.

Un sentito ringraziamento va alla Professoressa Chiara Braghin per la preziosa guida e i consigli durante lo sviluppo di questa tesi, nonché a tutti coloro che hanno contribuito con il loro supporto e le loro idee. Ringrazio inoltre i miei colleghi e amici, che hanno offerto suggerimenti utili e stimolanti discussioni durante il processo di ricerca e scrittura. Infine, un ringraziamento speciale va alla mia famiglia, il cui supporto costante e incoraggiamento sono stati fondamentali per il raggiungimento di questo obiettivo.

Organizzazione della tesi

La tesi è organizzata come segue:

- nel Capitolo 1 viene presentato lo stato dell'arte, con una panoramica sui protocolli di sicurezza e sui metodi di autenticazione;
- nel Capitolo 2 viene approfondita la tematica della comunicazione sicura tra client e server, con un'attenzione particolare all'uso del tool OpenSSL.
- nel Capitolo 3 si analizzano le varie metodologie di autenticazione;
- nel Capitolo 4 si trattano le problematiche e le soluzioni legate all'autorizzazione.

Capitolo 1

Stato dell'Arte

Conoscere lo stato dell'arte è fondamentale per comprendere il contesto e le basi teoriche della sicurezza dei sistemi web moderni. In questo capitolo, verranno esaminate le tecnologie e i protocolli attualmente in uso per garantire la sicurezza delle comunicazioni e delle autenticazioni nel web.
Si esplorerà anche la distribuzione e l'adozione dei moderni server web, che giocano un ruolo cruciale nella gestione della sicurezza delle applicazioni web. Server come Apache, Nginx e quelli basati su Java offrono funzionalità avanzate di sicurezza che saranno analizzate in dettaglio.
Infine, verranno presentate le capacità di sicurezza del linguaggio di programmazione Java, spesso utilizzato per sviluppare applicazioni server-side, e il modo in cui queste capacità sono integrate e sfruttate nei server web moderni.
L'obiettivo di questo capitolo è fornire una panoramica completa delle tecnologie e delle pratiche attuali nel campo della sicurezza web, preparare il lettore alla comprensione dei capitoli successivi e offrire una solida base teorica per l'implementazione delle soluzioni di sicurezza trattate nel resto della tesi.

1.1 Sicurezza dei protocolli di comunicazione: SSL e TLS

Lo stack protocollare TCP/IP di Internet non include di per sé funzionalità di sicurezza, a causa delle sue origini storiche in cui tali esigenze non erano prioritarie. Solo in seguito le questioni di sicurezza sono diventate sempre più rilevanti, portando allo sviluppo di nuove versioni dei protocolli con maggiore attenzione alla sicurezza. La prima implementazione che risolveva il problema di sicurezza relativo alle comunicazioni è stato il protocollo Secure Socket Layer (SSL) [1], che nella sua prima versione permetteva una cifratura simmetrica con una chiave a 40 bit. La lunghezza massima

della chiave è stata imposta dal governo statunitense per motivi di "sicurezza nazionale".
Le implementazioni moderne utilizzano chiavi per la cifratura simmetrica a 128 (o più) bit. Attualmente le versioni v2 e v3 di SSL sono considerate insicure in quanto utilizzano l'algoritmo MD5, ormai obsoleto, per la generazione di MAC. Il protocollo Transport Layer Security (TLS) è il successore di SSL. Di TLS vengono utilizzate le versioni TLS 1.2 [2] e TLS 1.3 [3]. Il protocollo TLS permette a due endpoint di comunicare attraverso una rete in modo tale da prevenire la falsificazione, la manomissione e l'intercettazione dei dati scambiati.
Nel contensto moderno, TLS è utilizzato soprattutto nella messa in sicurezza del canale instaurato tra un browser e un Server Web, dove le proprietà della triade CIA (Confidenzialità, Integrità e Disponibilità) devono essere soddisfatte.
Il protocollo definisce anche una modalità di autenticazione, la quale può essere: unilaterale, chiamata TLS 1-way, nel caso in cui solo il Server Web si autentica (use case più comune), oppure bilaterale, chiamata TLS 2-way, nel caso in cui entrambe le parti si autenticano reciprocamente. Le modalità di autenticazione definite fanno utilizzo delle proprietà della cifratura asimmetrica, incorporando nei messaggi scambiati i cosiddetti "certificati", che vengono rilasciati da delle autorità di certificazione (in inglese, Certification Authority) esterne.
La scelta strategica della IEEE di lasciare un'alta flessibilità sulla scelta dei diversi algoritmi crittografici che possono essere combinati per ottenere le proprietà di sicurezza desiderate fa sì che le *CipherSuite* definibili in TLS possano rispondere a esigenze di compatibilità, prestazioni, robustezza e adattabilità personalizzate. Inoltre permettono di definire dei metodi di paragone sulla quale è possibile eseguire una serie di test per capire quale *CipherSuite* sia più adatta al contesto in cui ci si trova. Allo stato attuale, secondo uno studio condotto da Scott Helme [4], dove si testa un milione di Web Server online, le cinque *CipherSuite* di default più utilizzate sono:

```
TLS_AES_256_GCM_SHA384
ECDHE-RSA-AES256-GCM-SHA384
ECDHE-RSA-AES128-GCM-SHA256
TLS_AES_128_GCM_SHA256
ECDHE-RSA-AES256-SHA384
```

L'utilizzo del protocollo TLS, soprattutto nelle versioni precedenti alla 1.3, non garantisce la copertura completa della superficie di attacco nel caso in cui non vengano prese in considerazione ulteriori raccomandazioni. Nel documento rilasciato dall'Agenzia per l'Italia Digitale (Agid) con nome "Raccomandazioni Agid in merito allo standard Transport Layer Security (TLS)" [5], si stabiliscono ulteriori controlli in merito ad attacchi noti al protocollo, in particolare:

- La rinegoziazione di una sessione TLS è vulnerabile ad attacchi noti, per questo motivo si DEVE rifiutare la rinegoziazione della sessione iniziata dal client.
- Analogamente, se viene usato HTTP/2 over TLS 1.3 non DEVE essere abilitata la Post-Handshake Authentication.
- Tutte le versioni precedenti alla 1.3 DEVONO disablitare la compressione TLS per evitare attacchi diretti ad essa come il noto CRIME [6].
- L'utilizzo dell'estensione Heartbeat di TLS DEVE essere disabilitata per evitare attacchi diretti ad essa come il noto HEARTBLEED [7].

1.2 Metodi di Autenticazione forniti da HTTP

L'Autenticazione legata al protocollo HTTP ha visto una significativa evoluzione negli ultimi decenni. Siamo passati da dei meccanismi di autenticazione semplici e insicuri a soluzioni complesse che fanno utilizzo di primitive crittografiche avanzate.
La prima implementazione sul mercato di un meccanismo di autenticazione si ha con l'introduzione del protocollo HTTP/1.0 nel 1966. L'obiettivo era quello di fornire un metodo semplice e standardizzato per autenticare gli utenti nel web, senza la necessità di meccanismi di sicurezza complessi.

1.2.1 Basic Authentication

La prima forma di autenticazione viene chiamata *Basic Authentication*. Essa viene ancora utilizzata in applicazioni web semplici ed è necessario combinarla con TLS per evitare che le credenziali siano trasmesse in chiaro sulla rete.
Sostanzialmente, le richieste a risorse (pagine) protette dalla Basic Authentication rispondono con uno stato `401 Unauthorized` e un header `WWW-Authenticate` che specifica il metodo di autenticazione richiesto (in questo caso `Basic`). A questo punto, il client risponde con un header `Authorization` che include le credenziali dell'utente codificate in Base64.
L'utilizzo della Basic Authentication presentava numerosi problemi di sicurezza:

- Le credenziali potevano essere decodificate da chi si inseriva nel canale usando banalmente un decoder Base64.
- Le credenziali potevano essere modificate da chi si inseriva nel canale e la manomissione non veniva identificata.
- Le credenziali potevano essere inviate ad un falso Server Web in quanto esso non si autenticava.

1.2.2 Digest Authentication

Visti i numerosi problemi della Basic Authentication, nel 1997, è stata introdotta la *Digest Authentication* come parte integrante del protocollo HTTP/1.1. La Digest Authentication, con l'utilizzo di funzioni di hashing e nonce, promette una soluzione più sicura e robusta, pur mantenendo una relativa semplicità di implementazione rispetto a metodi di autenticazione più complessi.
Sostanzialmente, nel caso venga fatta una richiesta di risorse (pagine) protette dalla Digest Authentication, il server risponde con uno stato `401 Unauthorized` ed un header `WWW-Authenticate` che specifica il metodo di autenticazione richiesto (`Digest`) ed un numero casuale (nonce). A questo punto, il client risponde con un header `Authorization` che include l'hash generato dalle credenziali del client concatenate al nonce. La verifica fatta dal Server è il confronto tra l'hash ricevuto e l'hash calcolato conoscendo le credenziali dell'utente.
L'utilizzo della Digest Authentication presentava anch'essa dei problemi:

- Anche se i nonce sono progettati per prevenire i replay attack, questo metodo risulta comunque vulnerabile al Man in the Middle sopratutto nei casi in cui non viene usato TLS.
- Le funzioni di hashing utilizzate erano deboli e generavano collisioni molto frequentemente. Questo permetteva di autenticarsi anche senza conoscere le credenziali dell'utente.
- A livello computazionale, ogni richiesta richiedeva il calcolo di hash sia lato client che lato server il quale che generava overhead sulle prestazioni.

La Digest Authentication viene ancora oggi utilizzata, ma è comunque meno comune rispetto ad alcuni metodi più moderni. Inoltre, a causa delle debolezze delle funzioni di hashing utilizzate, si preferisce utilizzare la Basic Authentication over HTTPS.

1.3 Metodi di Autenticazione moderni

I precedenti metodi di autenticazione vengono raccolti nella categoria delle autenticazioni basate sulla password. Questa categoria è la prima nata (circa negli anni 60) e consiste nella conoscenza, da parte dell'utente, di un segreto.
I metodi di autenticazione basati su password presentano diversi problemi e limitazioni, che ne compromettono l'efficacia e la sicurezza. Basti pensare alle vulnerabilità legate alla loro debolezza, alla loro gestione e al loro riutilizzo da parte dell'utente.
Per questo durante gli anni si è cercato di progettare sistemi di autenticazione che non prevedono l'uso di password, i metodi così detti: "Passwordless".
Nei primi anni 90 si sperimenta l'utilizzo della Biometria per autenticare gli utenti,

così si iniziano a progettare dispositivi in grado di registrare e verificare le caratteristiche biometriche dell'uomo come: le impronte digitali, le iridi o il dna.
Verso i primi anni del nuovo millennio si scoprirono i vantaggi legati alla robustezza della combinazione di metodi di autenticazione diversi, come ad esempio l'utilizzo della Biometria (qualcosa che sei), combinata all'utilizzo di chiavette hardware OTP (qualcosa che hai). Nasce così la *2-Factor Authentication*, che prevedeva l'utilizzo di due metodi di autenticazione distinti e conseguentemente la *Multi-Factor Authentication*, se veniva combinata anche la conoscenza di una password (qualcosa che so).
Negli stessi anni nasce l'idea di centralizzare la gestione delle password in un unico componente presso il quale l'utente si poteva autenticare una sola volta e successivamente accedere a tutti i servizi a cui si è registrato senza effettuare nuovamente l'autenticazione. Questo use-case viene definito *Single Sign-On* (SSO) e cerca di contrastare i problemi legati all'utilizzo di password deboli o al loro riutilizzo per servizi diversi.

1.3.1 SAML 2.0

SAML 2.0 [8] è uno standard aperto per l'autenticazione e l'autorizzazione. La sua prima versione risale al novembre del 2002 e fu sancito come standard OASIS. Consente lo scambio sicuro di informazioni di autenticazione e autorizzazione tra entità diverse come specificato nel profilo "Web Browser SSO". È ampiamente utilizzato nelle applicazioni web aziendali ancora oggi.
Composto da due componenti principali:

- L'Identity Provider (IdP), ovvero l'entità che autentica l'utente e genera il token SAML contenente le asserzioni di autenticazione.
- Il Service Provider (SP), ovvero l'entità che richiede l'autenticazione dell'utente e riceve il token SAML dall'identity provider per autorizzare l'accesso.

Basato sulle asserzioni, ovvero dei documenti XML che contengono informazioni sull'utente, tra cui le dichiarazioni di autenticazione, attributi dell'utente e informazioni di autorizzazione, esso riesce ad essere uno strumento potente per le organizzazioni che cercano di implementare soluzioni di Single Sign-On e gestione federata delle identità in ambienti complessi e distribuiti in modo sicuro, usabile, efficiente ed interoperabile.

1.3.2 Token JWT

La progressiva richiesta di implementare soluzioni "Passwordless" fa sì che nel 2007 inizi lo sviluppo delle prime forme di autenticazione basate sui token software, ovvero dei "gettoni" firmati dall'autorità che li rilascia e contenenti tutte le informazioni di

autenticazione dell'utente.
Nel 2015 vengono standardizzati i *JSON Web Token* (JWT) [9], strettamente legati ai metodi di autenticazione federata e Single Sign-On. Essi vengono classificati come leggeri, sicuri e facilmente utilizzabili in contesti distribuiti. Un token JWT è composto da:

- un *header* che contiene informazioni sul tipo di token e l'algoritmo di firma utilizzato;
- un *payload* che contiene le dichiarazioni (claim), che possono includere informazioni sull'utente e altre informazioni necessarie;
- la *firma* creata usando l'header e il payload insieme a una chiave segreta o a una coppia di chiavi pubblica/privata.

1.3.3 OpenID Connect

OpenID Connect [10] (OIDC) è un protocollo di autenticazione progettato per verificare l'identità degli utenti finali e ottenere informazioni di base sul loro profilo in modo sicuro. È stato sviluppato per fornire un meccanismo semplice e standardizzato per implementare l'autenticazione Single Sign-On nelle applicazioni web e mobili. Basato sul protocollo di Autorizzazione OAuth 2.0 [11], utilizza i token JWT per gestire le informazioni di autenticazione dell'utente.
Il provider OIDC deve mettere a disposizione degli endpoint REST necessari per lo use-case di autenticazione e autorizzazione, in particolare:

- *Authorization Endpoint*: Dove l'utente viene autenticato e concede i permessi;
- *Token Endpoint*: Dove l'applicazione ottiene i token di accesso;
- *UserInfo Endpoint*: Fornisce informazioni aggiuntive sull'utente, come il profilo utente e le informazioni di contatto.

Il protocollo supporta il flusso di autenticazione "Authorization Code Flow", principalmente utilizzato da applicazioni back-end e permette di gestire l'autenticazione con un Back-Channel instaurato tra SP e IdP senza l'interazione del browser dell'utente. Questo rende OIDC lo standard più sicuro per l'autenticazione SSO nel contesto mobile.

1.3.4 Passwordless Authentication con FIDO2

L'utilizzo dei token è comunque vulnerabile a una serie d'attacchi legati al loro rilascio, che richiede l'utilizzo di una password (anche se singola). Verso la fine degli

Anni 2010, la FIDO Alliance ha sviluppato lo standard FIDO2 [12] (Fast IDentity Online) che elimina completamente l'utilizzo di password e abbraccia il paradigma "Passwordless".
Il suo funzionamento permette l'accesso sicuro ai servizi online senza aver bisogno di Password o SMS OTP, ma mediante il solo utilizzo dello smartphone, del tablet o del PC. Da un'autenticazione che si basa su qualcosa che l'utente conosce si passa quindi a un'autenticazione basata su qualcosa che l'utente possiede. L'autenticazione FIDO, inoltre, usa le proprietà della crittografia a chiave pubblica.

1.3.5 Metodi di Autenticazione AI-Based e Blockchain-Based

La direzione del concetto di autenticazione è evoluta significativamente negli ultimi due anni. La continua crescita esponenziale dei sistemi di Intelligenza Artificiale e delle transazioni decentralizzate sta influenzando profondamente anche i sistemi di autenticazione.
Già dal 2010 era possibile osservare che alcune tecnologie di autenticazione basate su sistemi decentralizzati, come la *Blockchain*, fondavano il loro principio sul lasciare agli utenti la libertà di gestire le proprie informazioni personali, concetto noto come *Self-Sovereign Identity* (SSI).
Oggi, a questa tecnologia si affianca la cosiddetta "Autenticazione Adattiva", che migliora la sicurezza identificando anomalie e adattando i metodi di verifica in base ai rischi percepiti, grazie all'utilizzo dell'Intelligenza Artificiale.

1.4 Suddivisione mercato dei moderni Server Web

Una volta discusso lo stato dell'arte dei metodi di autenticazione, è necessario capire quali server web siano in grado di implementare tali protocolli. Secondo una statistica condotta da Meta DB e pubblicata sul loro sito web [13], al momento della stesura di questa tesi, le quote di mercato risultano così suddivise:

```
Apache          3.987.024 (37,35%)
Nginx           2.131.013 (19,96%)
Cloudflare        868.098 ( 8,13%)
GSE               588.026 ( 5,51%)
Microsoft IIS     531.937 ( 4,98%)
```

La maggior parte della torta è conquistata dai server web Apache, tra cui spiccano Apache Tomcat e Apache Geronimo, entrambi open-source e basati su Java. Inoltre, nell'ambito enterprise, troviamo frequentemente tecnologie proprietarie come Oracle WebLogic Server e RedHat JBoss Enterprise Application Platform, anch'esse sviluppate in Java. Da non dimenticare l'ampio utilizzo di Java nello sviluppo della maggior

parte delle applicazioni per dispositivi mobili su piattaforma Android.
Deduciamo quindi che l'utilizzo di Java è ampiamente diffuso sia in ambito web che mobile. È quindi necessario comprendere le capacità di sicurezza integrate nel linguaggio che hanno contribuito alla sua vasta adozione.

1.5 Capability di Sicurezza di Java utilizzate dai moderni Server-Web

1.5.1 Sicurezza del linguaggio Java e Bytecode Verifier

Il linguaggio Java è stato concepito per essere type-safe ovvero riesce a prevenire errori di tipo. Tra i controlli previsti è presente la gestione della memoria da liberare in modo automatico tramite garbage collection ed il controllo sui range degli array per prevenire problemi di Buffer Overflow.
Un Bytecode Verifier viene invocato per assicurarsi che solo il codice compilato legittimo sia eseguito dalla Java Virtual Machine, ovvero il componente virtuale che ha il compito di eseguire il codice in modo isolato. Esso controlla anche eventuali violazioni di memoria e typecast illegali.
Java mette a disposizione delle keyword di accesso che possono essere assegnate alle classi, ai metodi o agli attributi per permettere agli sviluppatori di filtrare l'accesso alle varie implementazioni:

- private, consente l'accesso solo a chi è dentro la classe.
- protected, permette l'accesso solo alle sottoclassi o alle classi dello stesso package.
- public, permette sempre l'accesso.

1.5.2 Security Provider

Il Java Development Kit (JDK) di Java definisce un insieme di API per accedere alle più conosciute funzioni di sicurezza. Queste API vengono implementate secondo i principi di indipendenza, interoperabilità e estensibilità, ed esse prendono forma attraverso i Security Provider.
Ogni Security Provider incapsula una lista di servizi di sicurezza già implementati e permette di accedervi attraverso delle interfacce standard. Ad esempio per richiedere il Provider MessageDigest che usa come algoritmo SHA-256 implementato da SunJSSE si usa:

```
MessageDigest md = MessageDigest.getInstance("SHA-256", "SunJSSE");
```

La JDK di default mette in ordine di preferenza le varie implementazioni fornite dai vari provider. Infatti nel caso in cui non venga specificato un Provider la JDK ci fornisce il servizio implementato dal Provider con priorità più alta.
Alcuni Provider possono essere estesi tramite una extend della classe oppure possiamo direttamente scaricare le classi Provider da terze parti.

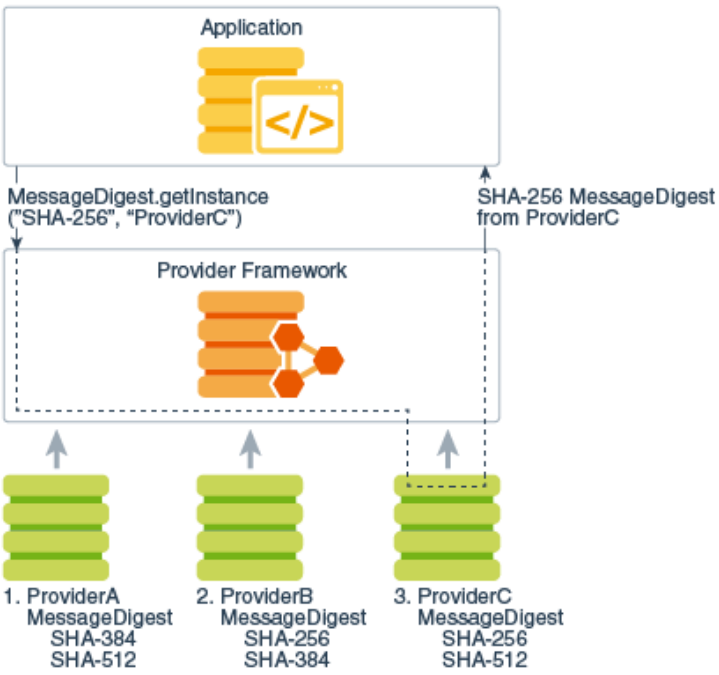

Figura 1: Richiesta dell'algoritmo di hashing SHA-256 al ProviderC

1.5.3 Java Criptography Architecture

La Java Criptography Architecture (JCA) è il framework di Java che mette a disposizione le funzionalità crittografiche più comuni. Questo framework è un insieme di API Provider-Based e permette di richiamare metodi che implementano: Algoritmi di Hashing, Algoritmi di Firma Digitale, Cifratura Simmetrica/Asimmetrica, Cifratura Password-Based, Algoritmi basati sulle curve ellittiche, Algoritmi di Key-Exchange, Generatori di numeri random, Generatori di coppie di chiavi e Message Authentication Codes.

1.5.4 Keytool e Key-Managers

Per implementare le funzioni relative alla Public Key Infrastructure, Java mette a disposizione una struttura su file, chiamata keystore, che permette lo storage di coppie di chiavi pubbliche/private e relativi certificati. La stessa struttura può essere utilizzata per salvare i certificati esterni fidati e in quel caso viene chiamata truststore. Il file cacerts è il truststore di default che contiene i certificati dei top level CA globali ed è presente (anche se con nomi diversi) in ogni sistema operativo.
Java permette di manipolare queste strutture dati tramite il suo tool proprietario Keytool, che opera su file con estensione .JKS (Java Key Stores). Esso permette la conversione di tali strutture nei formati standard compatibili anche con OpenSSL, ad esempio PKCS#12 [14].
Queste strutture dati sono importanti soprattutto durante l'autenticazione nel protocollo TLS e permettono la lettura e la verifica dei certificati scambiati.
Proviamo a generare una coppia di chiavi RSA a 2048 bit valida 90 giorni in KEYSTORE protetto dal passphrase PASS:

```
keytool -genkeypair -alias ALIAS -keyalg RSA \
    -keysize 2048 -validity 90 -storepass PASS -keystore KEYSTORE
```

Proviamo a esportare il certificato legato alla coppia di chiavi in KEYSTORE ed a salvarlo in un file cert.crt

```
keytool -exportcert -alias ALIAS -storepass PASS \
    --keystore KEYSTORE -rfc -file cert.crt
```

Proviamo ad importare il certificato cert.crt in TRUSTSTORE

```
keytool -importcert -alias ALIAS -storepass PASS \
    -keystore TRUSTSTORE -file cert.crt
```

1.5.5 Java Secure Socket Extension

In Java esiste una API che fornisce una interfaccia semplice per accedere a Socket che implementa i protocolli di sicurezza più comuni come SSL, TLS e DTLS tramite una classe che si chiama SSLSocket. Visto che SSLSocket è una sottoclasse di Socket essa permette l'utilizzo delle stesse interfacce esposte da Socket.
Utilizzando le classi KeyManager e TrustManager è possibile leggere i file creati con Keytool che contengono le chiavi e i certificati.

```
SSLContext ctx = SSLContext.getInstance("TLSv1.3");
KeyManagerFactory kmf = KeyManagerFactory.getInstance();
KeyStore ks = KeyStore.getInstance("JKS");
```

```
ks.load(new FileInputStream("s_keystore.jks", passphrase));
kmf.init(ks, passphrase);
ctx.init(kmf.getKeyManagers(), null, null);
```

1.5.6 Java Authentication and Authorization Service

Java espone delle API che permettono anche l'autenticazione dell'utente attraverso dei moduli di login sotto il framework JAAS. Le applicazioni possono chiamare la classe LoginContext che legge il file di configurazione che specifica le classi LoginModule da chiamare (che contengono l'implementazione del servizio di autenticazione).
Anche le classi LoginModule sono state concepite con il principio di indipendenza e quindi non è necessario scrivere l'implementazione manualmente ma è possibile scaricare un modulo già scritto da terze parti o fare riferimento direttamente ai moduli già disponibile nella JDK.
Il File di configurazione contiene tutte le entry che è possibile specificare quando un LoginContext viene istanziato. Ad esempio:

```
Sample {
    SampleLoginModule requires debug=true;
}
```

La classe SampleLoginModule viene istanziata con priorità required (è necessario che l'utente si autentichi con questo modulo di Login per autenticarsi definitivamente) nel caso in cui il LoginContext sia istanziato con Entry "Sample".
Al posto della SampleLoginModule, che è stata creata manualmente, è possibile inserire qualsiasi classe già disponibile nella JDK come ad esempio la JndiLoginModule. LoginModule linkata al Framework JAAS (docs.oracle.com) L'autorizzazione avviene subito dopo l'autenticazione avvenuta con successo e Java assegna la classe Subject all'utente autenticato. La classe rappresenta i soggetti in uno spazio di autenticazione (Realm), ed è popolata dalle identità associate chiamate Principals, grazie alle quali è possibile scrivere delle Policy che permettono o negano l'accesso alle risorse.
Ogni classe Subject ha associati attributi di sicurezza chiamati Credentials e possono spaziare da chiavi di cifratura fino ad arrivare a password personali hashate.
La JVM per evitare di eseguire delle operazioni privilegiate si impersonifica nel soggetto utilizzando i metodi doAs() e doAsPrivileged(). I privilegi sono assegnati attraverso il Policy File che contiene per ogni Principal le relative permission. Ad esempio:

```
grant Principal SamplePrincipal "readFileUser" {
    permission java.util.PropertyPermission "java.home", "read";
    permission java.util.PropertyPermission "user.home", "read";
    permission java.io.FilePermission "foo.txt", "read";
}
```

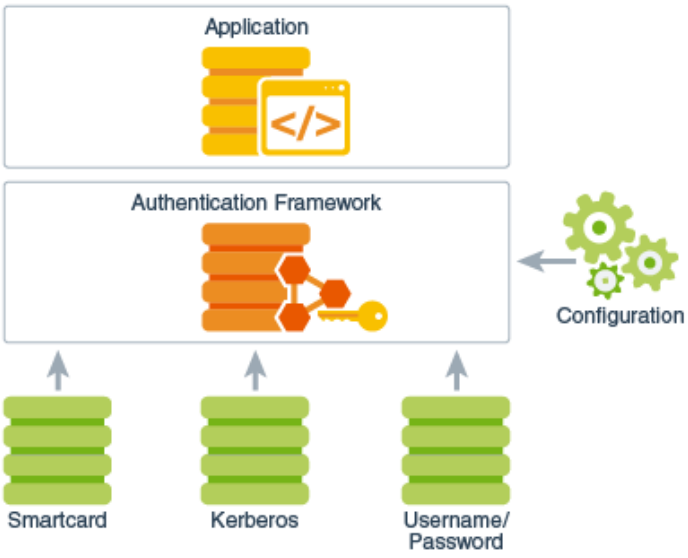

Figura 2: Plug-In dei moduli predefiniti nel framework di autenticazione

Il Principal "readFileUser" della classe SamplePrincipal ha assegnati i permessi per leggere: java.home, user.home e il file foo.txt.

1.6 Server Web Java-Based

Adesso che conosciamo le funzionalità di sicurezza che il linguaggio Java ci offre andiamo a vedere in quali Server Web esse vengono utilizzate (anche se nascoste al cliente). Prendiamo come esempio sia il Web Server Apache Tomcat, che rappresenta il lato open-source sia il Web Server Oracle WebLogic Server, che rappresenta il lato Enterprise.

1.6.1 Apache Tomcat

Tomcat è stato inizialmente sviluppato come parte del progetto JavaServer Web Development Kit (JSWDK) della Sun Microsystems. Nel 1999, Sun Microsystems ha donato il codice sorgente di Tomcat all'Apache Software Foundation, e da allora è stato sviluppato come progetto open-source sotto la ASF.
Apache Tomcat è noto per le sue numerose caratteristiche che lo rendono una scelta popolare per eseguire applicazioni web Java. Esso mette a disposizione 3 componenti :

- Catalina, ovvero il contenitore di servlet Java che permette la generazione di pagine web dinamiche.
- Coyote, ovvero il connettore HTTP che supporta il protocollo HTTP e ha il compito di ascoltare le connessioni in entrata su una determinata porta.
- Jasper, ovvero il motore JSP che analizza i file .jsp di back-end per compilarli in codice Java come servlet.

1.6.2 Oracle Weblogic

Weblogic è stato sviluppato dalla BEA System, poi acquisita da Oracle nel 2008. Noto per essere un application server Java Enterprise Edition , quindi incorpora delle componenti che solitamente sono richieste in applicativi di medie e grandi dimensioni. Alcune delle componenti principali sono:

- Java Messaging Standard (JMS) che permette alle componenti Java EE di creare, spedire e ricevere messaggi tramite delle API.
- JRockit JVM, ovvero una JVM che a differenza di quella standard è più indicata per applicazioni Java che hanno bisogno di caratteristiche di flessibilità, scalabilità, resilienza e gestione centralizzata.
- Oracle Coherence, ovvero una cache distribuita basata su Java in-memory che offre alta disponibilità, scalabilità e bassa latenza, throughput e performance per le applicazioni.

Inoltre , a differenza di Tomcat, mette a disposizione una console con interfaccia grafica accedibile tramite Browser dalla quale è possibile modificare le configurazioni dei vari domini creati.

1.7 Altri Server Web utilizzati

Nella prima posizione della statistica condotta da Meta DB sono presenti i Server Web Apache. In questa categoria troviamo anche: "Apache Http Server".

1.7.1 Apache Http Server

Apache HTTP Server è un server web open-source sviluppato dalla Apache Software Foundation nato nel 1995. E' la piattaforma modulare più diffusa, in quanto può operare su una grande varietà di sistemi operativi.

La sua modularità è data dal fatto che ad ogni richiesta del client vengono svolte funzioni specifiche da ogni modulo di cui è composto, come unità indipendenti. Questi moduli vengono specificati all'interno di un file di configurazione httpd.conf.
Dal punto di vista architetturale , l'operatività del servizio è gestita da un processo demone che legge il file di configurazione e permette l'accesso a uno o più siti. Il demone attraverso un ciclo di polling interroga continumente le linee di logica applicativa da cui possono pervenire messagi di richiesta. Le fasi del ciclo sono:

1. Translation: si traduce la richiesta del client.
2. Access Control: si controlla l'autorizzazione della richiesta.
3. MIME Type: si identifica il tipo di contenuto e si decide quali moduli servono la richiesta.
4. Response: si genera e si spedisce la risposta al client.
5. Logging: si tiene traccia dell'attività svolta.

I moduli di Apache HTTP Server permettono anche di configurare il Server in modalità proxy e permettono di specificare, quindi, tutti gli switch di controllo per il forwarding o per il load balancing.

1.7.2 Nginx

Al secondo posto della statistica di Meta DB troviamo Nginx. Nginx è un server web open-source leggero e ad alte prestazioni, concepito con lo scopo esplicito di superare le performance di Apache. Sviluppato nel 2002 da Sysoev , permette non solo l'esposizione di pagine web dinamiche o statiche ma anche la funzionalità di web proxy con bilanciamento del carico e supporto SSL.
E' proprio per questa seconda funzionalità che secondo il Web Server Survey Netcraft di marzo 2015 Nginx è stato riconosciuto come il terzo server web più utilizzato in tutti i domini e il secondo server web più utilizzato per tutti i siti "attivi" e la sua copertura è ancora oggi in aumento.
Anche nginx mette a disposizione un file di configurazione che permette la centralizzazione di tutte le opzioni che sono specificabili, ovvero nginx.conf.
Nginx offre anche un servizio a pagamento chiamato Nginx Plus che estende le funzionalità con moduli specifici per soluzioni Enterprise di medie e grandi dimensioni.

1.7.3 Node.js

Tra i server web più conosciuti ed utilizzati è presente anche un prodotto basato interamente su Javascript. Node.js è stato originariamente creato nel 2009 da Ryan

Dahl e supportato successivamente dal motore JavaScript V8 di Google. Adesso è portato avanti dalla Node.js Foundation.
Node.js è un sistema a runtime open-source orientato agli eventi, ciò vuol dire che Node richiede al sistema operativo di ricevere notifiche al verificarsi di determinati eventi. Questa proprietà è intrinseca nel linguaggio di programmazione Javascript, che originariamente era destinato esclusivamente al codice client-side. Il suo utilizzo è quindi destinato a coloro che vogliono sviluppare applicazioni web real-time in quanto ottimizzato per massimizzare il throughput e la scalabilità.
La sua semplicità di utilizzo, la velocità dei thread non bloccanti (async-await model), la pacchettizzazione dei moduli nell'ecosistema NPM (Node package manager), il suo supporto multipiattaforma e molte altre caratteristiche hanno fatto sì che la crescità della sua adozione per gli applicativi web sia stata esponenziale e che tutt'oggi risulti una soluzione all'avanguardia.

1.8 Identity and Access Management(IAM)

Nella terza sezione abbiamo discusso lo stato dell'Arte dei protocolli di autenticazione, tra i quali troviamo quelli per gestire lo use-case SSO. Nella sesta e settima abbiamo discusso quali Server Web possono essere utilizzati per l'erogazione di servizi dopo l'autenticazione dell'utente. Generalmente in questa metodologia si prevede l'esistenza di un punto centralizzato per gestire l'accesso degli utenti.
Le soluzioni IAM hanno il compito di garantire che le persone autorizzate di un'organizzazione (identità) possano accedere agli strumenti software necessari per svolgere il loro lavoro. Inoltre, consentono una gestione centralizzata delle policy di autorizzazione delle applicazioni dei dipendenti.
Oggi esistono diverse soluzioni che implementano il modello IAM e, secondo una ricerca condotta da Shubham Munde [15] e riportata sul sito marketresearchfuture.com, il loro mercato è destinato a crescere da 19.658,33 milioni di dollari nel 2024 fino a 71.650,50 milioni di dollari entro il 2032. Questa crescita è alimentata dal crescente bisogno di soluzioni di sicurezza robuste. Tra le aziende che sviluppano queste soluzioni troviamo IBM, Oracle, Okta, Ping Identity, SAP, CyberArk, AWS e WSO2.
In questa sezione mi concentro su WSO2, poiché durante il mio tirocinio curricolare in azienda mi è stato richiesto di studiarla, essendo utilizzata da alcuni clienti esterni appartenenti alla Pubblica Amministrazione Italiana.

1.8.1 WSO2

WSO2 è una tecnologia open-source fondata nel 2005. Distribuisce soluzioni cloud e on-premises per lo sviluppo di applicativi software, per la gestione centralizzata di API e recentemente ha integrato una soluzione IAM. La soluzione integrata IAM

si chiama "WSO2 Identity Server" ed è quella su cui ci concentreremo nei prossimi capitoli per implementare un Identity Provider (IdP) nello use-case di SSO.
Possiamo raggruppare le sue caratteristiche in:

- soluzione open-source sotto la licenza Apache 2 che permette trasparenza.
- soluzione scalabile fino a milioni di utenti senza incorrere in costi aggiuntivi.
- supporto alla maggior parte di protocolli di SSO.

Capitolo 2

Comunicazione sicura tra client e server

La trasmissione di dati sensibili su reti pubbliche esponse le informazioni a varie minacce, inclusi intercettazioni e attacchi man-in-the-middle, per questo la sicurezza delle comunicazioni è diventata una priorità assoluta. Per mitigare questi rischi sono nati diversi protocolli che garantiscono la confidenzialità, l'integrità e l'autenticità delle comunicazioni.
Questo capitolo esamina in modo approfondito i metodi utilizzati per assicurare che le comunicazioni tra client e server avvengano in modo sicuro. In particolare, ci concentreremo sullo strumento OpenSSL, uno dei toolkit più diffusi per la crittografia e per la gestione delle chiavi. Inoltre, esploreremo le configurazioni dei canali TLS instaurabili evidenziando i casi d'uso e i vantaggi di ciascuna configurazione.
L'obiettivo è fornire una metodologia chiara e comprensibile per implementare e gestire canali di comunicazione sicuri in un'architettura web moderna. La comprensione e l'adozione di questi protocolli di sicurezza sono fondamentali per garantire la fiducia degli utenti e la conformità alle normative sulla protezione dei dati.

2.1 Panoramica sul tool OpenSSL

Il progetto OpenSSL, nato nel 1998, rappresenta la principale implemtazione open-source del protocollo SSL/TLS. La libreria principale è scritta in C e supporta un'ampia gamma di primitive crittografiche.
OpenSSL offre strumenti per testare il funzionamento di un applicativo che si basa sullo stack TLS. Queste funzionalità rivestono particolare importanza durante le fasi di troubleshooting e nel Proof of Concept (PoC).
In particolare ci concentreremo sull'uso del tool OpenSSL per il processo di "Certificate Issuance", ovvero quel processo che permette ad un client di farsi rilasciare un

certificato da un Server Autorevole (CA) che attesta il possesso di una chiave pubblica.
Il processo è così suddiviso:

1. Generazione della coppia di chiavi pubblica/privata.
2. Creazione e invio della Certificate Signing Request ad una CA.
3. Verifica e generazione del certificato da parte della CA.

2.1.1 Generazione della coppia di chiavi pubblica/privata

Analogamente a Keytool anche OpenSSL permette la generazione di chiavi tramite il comando "genpkey". Gli switch più importanti sono quelli che permettono di specificare l'algoritmo e le opzioni relative all'algoritmo scelto:

```
openssl genpkey out fd.key \
    -algorithm RSA -pkeyopt rsa_keygen_bits:2048
```

In questo modo viene creata una chiave privata RSA a 2048 bit. Aggiungendo lo switch "-aes-128-cbc" cifriamo la chiave con una password usando AES a 128bit (noto algoritmo di cifratura simmetrica).
A questo punto la chiave creata è in formato standard PKCS#8 [16] e per visualizzarla in OpenSSL si usano gli switch "-in -text -noout".

2.1.2 Creazione di una Certificate Signing Request (CSR)

Una volta creata una chiave privata è necessario creare una Richiesta di certificazione firmata (CSR). La CSR è una richiesta formale ad una CA di farsi generare un certificato a partire da una chiave pubblica:

```
openssl req -new -key fd.key -out fd.csr
```

Il comando è interattivo e verranno richieste ulteriori informazioni per la generazione della CSR (ad esempio il Subject).
Anche se con "-key" viene passato il file che contiene la coppia di chiavi, OpenSSL estrae solo quella pubblica. Il file generato è un .csr ed è in formato PKCS#10 [17] e può essere visualizzato anch'esso con i soliti switch specificati sulla generazione della coppia di chiavi.

2.1.3 Verifica e generazione del certificato da parte della CA

Quando la CSR raggiunge la CA, quest'ultima deve verificare che sia firmata correttamente e nel caso di esito positivo essa deve generare il relativo certificato richiesto. Alcune volte è possibile generare un certificato self-signed ovvero firmato da chi ha generato anche la CSR. Ad esempio in OpenSSL si usa:

```
openssl x509 -req -days 365 -in f.csr /
    -signkey fd.key -out fd.crt
```

Si specifica i giorni di validità del certificato, la CSR e la chiave privata con la quale firmare il certificato. Inoltre nelle ultime versioni dei certificati X.509 [18] è possibile specificare delle estensioni, come il Subject Alternative Name, utilizzato per l'identificazione del richiedente, usando lo switch "-extfile" e specificando un file contenente le estensioni da includere.

2.2 Guida alla creazione di una CA

Oltre alle funzioni di base per la gestione, la creazione e la modifica delle chiavi e dei certificati, OpenSSL permette di simulare il comportamento di un'autorità di certificazione (CA) utilizzando un file di configurazione.
In questa sezione, ci concentreremo sulla configurazione di una CA, che verrà poi utilizzata nei prossimi capitoli per istanziare Server CA privati e pubblici pronti all'uso.

2.2.1 Configurazione della Root CA

La configurazione è un file .conf che contiene le direttive necessarie a OpenSSL per il suo funzionamento.
La prima parte del file specifica le informazioni di base ed è divisa in due sezioni: "default" e "ca_dn":

```
[default]
name                = root-ca
domain_suffix       = example.com
aia_url             = http://$name.$domain_suffix/$name.crt
crl_url             = http://$name.$domain_suffix/$name.crl
ocsp_url            = http://ocsp.$name.$domain_suffix:9080
default_ca          = ca_default
name_opt            = utf8,esc_ctrl,multiline,lname,align
```

```
[ca_dn]
countryName         = "IT"
organizationName    = "Example"
commonName          = "Root CA"
```

Nella sezione [default] si definiscono le impostazioni di base, come il nome della CA e il suffisso di dominio.
Si specificano anche gli URL per accedere al certificato della CA, alla lista di revoca dei certificati(CRL) e al servizio OCSP [19], che serviranno rispettivamente per verificare l'autenticità del certificato, controllare la revoca e verificare lo stato dei certificati.
Inoltre, si indica il profilo di CA di default da utilizzare, le opzioni per la visualizzazione dei nomi nei certificati, come l'uso della codifica UTF-8, l'escape dei caratteri di controllo, la visualizzazione su più righe, l'uso dei nomi abbreviati e l'allineamento dei nomi.
La sezione [ca_dn] definisce il "Distinguished Name" (DN) per la CA, che è un identificativo unico.
La seconda parte specifica il collocamento dei file utilizzati da OpenSSL e la definizione di una policy di matching:

```
[ca_default]
home                = .
database            = $home/db/index
serial              = $home/db/serial
crlnumber           = $home/db/crlnumber
certificate         = $home/$name.crt
private_key         = $home/private/$name.key
RANDFILE            = $home/private/random
new_certs_dir       = $home/certs
unique_subject      = no
copy_extensions     = none
default_days        = 3650
default_crl_days    = 365
default_md          = sha256
policy              = policy_c_o_match

[policy_c_o_match]
countryName             = match
stateOrProvinceName     = optional
organizationName        = match
organizationalUnitName  = optional
```

```
commonName              = supplied
emailAddress            = optional
```

La CA utilizza come directory di lavoro la directory corrente. Il file "database" è un file di indice che tiene traccia dei certificati emessi. I file "serial" e "crlnumber" contengono rispettivamente il numero di serie dei certificati emessi e il numero di serie della lista di revoca. Il file "RANDFILE" memorizza dati casuali necessari per operazioni crittografiche. La directory certs è il contenitore dei certificati emessi. La CA è configurata per non imporre l'unicità dei soggetti nei certificati emessi e per non copiare estensioni dai certificati di richiesta. I certificati emessi avranno una durata di default di 3650 giorni, mentre le liste di revoca saranno valide per 365 giorni. L'algoritmo di hash predefinito utilizzato è SHA-256.
Nella seconda sezione si specifica una politica per il rilascio dei certificati che effettua dei controlli sugli attributi passati nella CSR generata dal client.
La terza parte contiene il comportamento della CA per generare le CSR. Sostanzialmente questa configurazione verrà utilizzata solo per la creazione del certificato di root CA il quale sarà self-signed:

```
[req]
default_bits        = 4096
encrypt_key         = yes
default_md          = sha256
utf8                = yes
string_mask         = utf8only
prompt              = no
distinguished_name  = ca_dn
req_extensions      = ca_ext

[ca_ext]
basicConstraints        = critical,CA:true
keyUsage                = critical,keyCertSign,cRLSign
subjectKeyIdentifier    = hash
```

Nella sezione [req] si definiscono le impostazioni generali per la creazione delle richieste di certificato, impostando la lunghezza della chiave a 4096 bit, abilitando la crittografia della chiave privata e specificando l'algoritmo di hash predefinito come SHA-256. Si abilità l'uso della codifica UTF-8. Si specifica che il nome distinto (DN) deve essere preso dalla sezione [ca_dn]. Inoltre, le estensioni per la richiesta di certificato devono essere prese dalla sezione [ca_ext].
Nella sezione [ca_ext] si definiscono le estensioni specifiche per i certificati della CA, impostando il certificato come critico e indicando che si tratta di una CA. La chiave associata può essere utilizzata per firmare certificati e liste di revoca. Infine si crea

un identificatore univoco per la chiave pubblica del soggetto utilizzando un hash.
La quarta e ultima parte del file di configurazione contiene le direttive che specificano i vincoli alle CA subordinate (sub-CA). Questa parte contiene anche i collegamenti che vengono aggiunti ai certificati rilasciati per controllare la relativa CRL e il relativo servizio OCSP:

```
[sub_ca_ext]
authorityInfoAccess         = @issuer_info
authorityKeyIdentifier      = keyid:always
basicConstraints            = critical,CA:true,pathlen:0
crlDistributionPoints       = @crl_info
extendedKeyUsage            = clientAuth, serverAuth
keyUsage                    = critical, keyCertSign, cRLSign
nameConstraints             = @name_constraints
subjectKeyIdentifier        = hash

[crl_info]
URI.0                       = $crl_url

[issuer_info]
caIssuer;URI.0              = $aia_url
OCSP;URI.0                  = $ocsp_url

[name_constraints]
permitted;DNS.0=example.com
permitted;DNS.1=example.org
excluded;IP.0=0.0.0.0/0.0.0.0
excluded;IP.1=0:0:0:0:0:0:0:0/0:0:0:0:0:0:0:0
```

Nella sezione [sub_ca_ext] vengono definite le estensioni specifiche per i certificati della sottoscrizione CA, con authorityInfoAccess che punta alla sezione [issuer_info].
L'identificatore della chiave dell'autorità viene sempre incluso tramite authorityKeyIdentifier, mentre basicConstraints imposta il certificato come critico, indicando che è una CA con un limite di percorso di certificazione (pathlen) pari a zero (quindi i sub-CA non possono avere altri sub-CA). Le estensioni di utilizzo della chiave sono configurate per autenticazione client e server con extendedKeyUsage, oltre a keyUsage che è impostato come critico, includendo le autorizzazioni per firmare certificati e liste di revoca. Le sezioni [crl_info] e [issuer_info] specificano i punti di distribuzione della CRL e dell OCSP Responder. Infine [name_constraints] definisce quali domini possono farsi generare un certificato.

2.2.2 Creazione directory Tree della Root CA

OpenSSL richiede che il server sia configurato utilizzando una struttura specifica per posizionare i file di configurazione, il database e i certificati rilasciati.
La directory deve seguire le posizioni definite nel file .conf della CA quindi:

```
$ mkdir root-ca
$ cd root-ca
$ mkdir certs db private
$ chmod 700 private
$ touch db/index
$ openssl rand -hex 16 > db/serial
$ echo 1001 > db/crlnumber
```

La sottodirectory certs fa lo storage dei certificati rilasciati e viene utilizzata per revocare un certificato e per tenerne traccia.
La sottodirectory db è il database usato da OpenSSL e traccia i numeri seriali per la creazione della lista CRL.
La sottodirectory private fa lo storage della chiave privata del Root-CA e del responder OCSP. Infatti si impostano i permessi per la lettura, la scrittura e l'esecuzione solo all'owner.

2.2.3 Generazione chiavi e certificato Root CA

Il file di configurazione creato verrà utilizzato per la generazione della coppia di chiavi pubblica/privata con la sezione "req" e per la generazione del certificato con la sezione "ca_ext":

```
$ openssl req -new -config root-ca.conf \
    -out root-ca.csr -keyout private/root-ca.key
$ openssl ca -selfsign -config root-ca.conf \
    -in root-ca.csr -out root-ca.crt -extensions ca_ext
```

2.2.4 Generazione CRL

La generazione della CRL è necessaria in quanto i nodi scaricando questa lista possono controllare se i certificati che ricevono sono stati revocati o no. In OpenSSL si usa lo switch "-gencrl":

```
$ openssl ca -gencrl -config root-ca.conf \
    -out root-ca.crl
```

2.2.5 Avviare il servizio OCSP Responder

L'Online Certificate Status Protocol (OCSP) è un'alternativa alle liste CRL, e consiste in un processo in ascolto che risponde alle richieste di controllo validità di un certificato in real time.
Come per la Root-CA è necessario generare una coppia di chiavi e un certificato. Infine sarà necessario far partire il servizio assegnandogli una porta per il listener:

```
$ openssl req -new -newkey rsa:2048 \
    -subj "/C=IT/O=Example/CN=OCSP Root Responder" \
    -keyout private/root-ocsp.key -out root-ocsp.csr
$ openssl ca -config root-ca.conf -in root-ocsp.csr \
    -out root-ocsp.crt -extensions ocsp_ext -days 30
$ openssl ocsp -port 9080 -index db/index \
    -rsigner root-ocsp.crt -rkey private/root-ocsp.key \
    -CA root-ca.crt -text
```

2.2.6 Configurazione della Subordinate CA

La Subordinate CA è una CA che risponde direttamente alla Root CA e ha dei vincoli specificati nella configurazione della Root CA. Spesso viene utilizzata per ricostruire correttamente la catena di CA nella verifica di un certificato per risalire al Top-level CA di cui il client si fida. Nel nostro caso la sub CA ha il compito di rilasciare i certificati ai Server e ai client della rete.
Anche la sua configurazione è definita tramite un file .conf, e partendo dal file root-ca.conf è possibile modificare solo alcune direttive per avere una Sub CA up and running.
Nome e Distinguished Name vanno cambiati come la porta dell'URL del servizio OCSP (ad esempio da 9080 a 9081). Nel [ca_default] modificare il copy_extensions a copy, in questo modo le estensioni specificate nella CSR sono copiate nel certificato quando verrà generato. Ovviamente con la policy definiamo quali veranno copiate.
E' possibile modificare la validità dei certificati e la rigenerazione della CRL.
Dobbiamo aggiungere due nuove sezioni utilizzate per le estensioni da aggiungere nei certificati rilasciati per i server e per i client :

```
[server_ext]
authorityInfoAccess     = @issuer_info
authorityKeyIdentifier  = keyid:always
basicConstraints        = critical,CA:false
crlDistributionPoints   = @crl_info
extendedKeyUsage        = clientAuth,serverAuth
keyUsage                = critical,digitalSignature,keyEncipherment
```

```
subjectKeyIdentifier    = hash

[client_ext]
authorityInfoAccess     = @issuer_info
authorityKeyIdentifier  = keyid:always
basicConstraints        = critical,CA:false
crlDistributionPoints   = @crl_info
extendedKeyUsage        = clientAuth
keyUsage                = critical,digitalSignature
subjectKeyIdentifier    = hash
```

Notare come nel basicConstraints si setta CA:false in quanto il certificato non è destinato ad una CA. La differenza tra i certificati rilasciati per il client e quelli per il server viene definita con l'uso dell'extendedKeyUsage e del keyUsage.

2.2.7 Generazione chiavi e certificato della Subordinate CA

Per la generazione della coppia di chiavi si usa il file di configurazione della sub CA ma per la generazione del suo certificato è necessaria la firma della Root CA e quindi il suo file di configurazione(non selfsign come nel caso della Root CA):

```
$ openssl req -new -config sub-ca.conf \
    -out sub-ca.csr -keyout private/sub-ca.key
$ openssl ca -config root-ca.conf -in sub-ca.csr \
    -out sub-ca.crt -extensions sub_ca_ext
```

2.2.8 Rilascio di un certificato

Adesso che la sub CA è pronta per essere utilizzata dobbiamo definire le operazioni per il rilascio alle parti del certificato data una CSR. Le operazioni saranno quindi diverse a seconda del ruolo della parte:

```
$ openssl ca -config sub-ca.conf -in server.csr \
    -out server.crt -extensions server_ext
$openssl ca -config sub-ca.conf -in client.csr \
    -out client.csr -extensions client_ext
```

Il primo comando e destinato al rilascio di un certificato ad un server della rete mentre il secondo è destinato al rilascio di un certificato ad un client della rete.

2.3 Instaurazione del canale TLS

Adesso vediamo come, attraverso raffinamenti successivi, instaurare tra un client e un webserver una connessione TLS 2-way utilizzando certificati rilasciati da una CA privata.
I diversi raffinamenti sono individuati da livelli ordinati:

1. Connessione tra client e webserver in chiaro.
2. Instaurazione del canale sicuro TLS 1-way tra client e webserver con certificati self-signed.
3. Instaurazione del canale sicuro TLS 1-way tra client e webserver con certificato server rilasciato da una CA privata.
4. Instaurazione del canale sicuro TLS 2-way tra client e webserver con certificati client e server rilasciati da una CA privata.

L'implementazione verrà eseguita sui Server Web menzionati nel capitolo sullo Stato dell'Arte e vedremo come le diverse tecnologie possono raggiungere lo stesso obiettivo in modi diversi. Inoltre la CA privata utilizzata sarà un'istanza della configurazione presentata nella precedente sezione.

2.3.1 Connessione tra client e webserver in chiaro

In questa sezione presento le varie configurazioni per configurare una semplice pagina web che restituisce HelloWorld. Per alcuni webserver si utilizza un file di configurazione mentre per altri è necessario l'interfacciamento con una console online.

Apache HTTP Server

Creazione del file html nella cartella htdocs:

```
#echo '<html><body><h1>Hello World!</h1></body></html>' \
# > htdocs/helloworld.html
```

Tomcat

Creazione webapp HelloWorld e creazione del file html nella cartella HelloWorld:

```
#cd webapps && mkdir HelloWorld
#echo '<html><body><h1>Hello World!</h1></body></html>' \
# > HelloWorld/index.html
```

NodeJs

Creazione file di routing app.js:

```
var http = require("http");
listener = function (req, res) {
    res.writeHead(200, {'Content-Type': 'text/html'});
    res.end(
        '<html><body><h1>Hello World!</h1></body></html>'
    );
}
server = http.createServer(listener);
server.listen(80);
```

Nginx

Aggiunta di un oggetto server nell'oggetto http nel file nginx.conf:

```
server {
    listen  80;
    location / {
        root    /usr/share/nginx/html;
        index   helloworld.html;
    }
}
```

Creazione del file html nella cartella root di nginx:

```
#echo '<html><body><h1>Hello World!</h1></body></html>' \
# > /usr/share/nginx/html/helloworld.html
```

Java WebLogic Server

Creazione di un file war modificando i file importati da una repo che contiene una simple-webapp:

```
#mkdir simple-webapp
#cd simple-webapp/
#mkdir WEB-INF
#curl https://raw.githubusercontent.com/.../web.xml > web.xml
#mv web.xml WEB-INF/
#echo '<html><body><h1>Hello World!</h1></body></html>' \
# > index.jsp
#jar -cvf simple-webapp.war *
```

Il file war, che contiene la webapp, può essere deployato utilizzando la console di weblogic. La guida per fare ciò è disponibile sul sito di oracle: docs.oracle.com.

2.3.2 Instaurazione del canale sicuro TLS 1-way tra client e webserver con certificati self-signed.

In questa sezione presento le varie aggiunte ai file di configurazione creati precedentemente. Inoltre è presente una sottosezione che presenta la creazione delle chiavi e dei relativi certificati da presentare ai browser. Essendo self-signed i browser visualizzerano un messaggio di errore che possiamo tranquillamente ignorare (Dal prossimo livello non avremmo più questo problema). Infine facciamo delle considerazioni sulle Ciphersuites di default utilizzate.

Keytool per Tomcat e WebLogic

Visto che Tomcat e WebLogic hanno bisogno della configurazione dei keystore per instaurare il canale TLS si utilizza keytool:

```
#keytool -genkey -alias server_key -keyalg RSA
```

Questo comando genera un keystore con coppia di chiavi pubblica/privata identificate dall'alias server_key con algoritmo RSA.

Openssl per Apache, Node e Nginx

Visto che Apache, Node e Nginx hanno bisogno di chiavi e certificati in formato PKCS#12 per instaurare il canale TLS si utilizza il tool openssl:

```
#openssl genpkey -out server.key -algorithm RSA \
# -pkeyopt rsa_keygen_bits:2048
#openssl req -new -key server.key -out server.csr
#openssl x509 -req -days 365 -in server.csr \
# -signkey server.key -out server.crt
```

Questo comando genera una coppia di chiavi privata/pubblica e un certificato self-signed relativo alla coppia di chiavi.

Apache HTTP Server

A differenza del precedente livello adesso Apache ci richiede di andare ad aggiungere un VirtualHost nel file httpd.conf:

```
LoadModule ssl_module modules/mod_ssl.so
Listen 443
<VirtualHost *:443>
    ServerName www.helloworld.com
    SSLEngine on
    SSLCertificateFile "/usr/local/apache2/keys/server.crt"
    SSLCertificateKeyFile "/usr/local/apache2/keys/server.key"
</VirtualHost>
```

Tomcat

Tomcat ci richiede di andare ad aggiungere un Connector nel file server.xml:

```
<Connector protocol=
        "org.apache.coyote.http11.Http11NioProtocol"
    sslImplementationName=
        "org.apache.tomcat.util.net.jsse.JSSEImplementation"
    port="443" maxThreads="200"
    maxParameterCount="1000"
    scheme="https" secure="true" SSLEnabled="true"
    sslProtocol="TLS">
    <SSLHostConfig>
        <Certificate
            certificateKeystoreFile=
                "/root/.keystore"
            certificateKeystorePassword=
                "passphrase"
            type="RSA"
        />
    </SSLHostConfig>
</Connector>
```

NodeJs

Nel caso di Node JS invece le modifiche saranno fatte nel file app.js, che diventerà:

```
const express = require('express');
const https = require('https');
const path = require('path');
const fs = require('fs');
const app = express();
app.use('/', (req, res, next) => {
```

```
    res.writeHead(200, {'Content-Type': 'text/html'});
    res.end(
        '<html><body><h1>Hello World</h1></body></html>'
    );
});
const server = https.createServer({
    key: fs.readFileSync(
        path.join(__dirname, 'keys', 'server.key')
    ),
    cert: fs.readFileSync(
        path.join(__dirname, 'keys', 'server.crt')
    ),
}, app);
server.listen(443, () => console.log('Listen on 443'));
```

Nginx

In Nginx sarà necessario modificare l'oggetto server, che diventerà:

```
server {
    listen  443 ssl;
    server_name www.helloworld.com;
    keepalive_timeout   70;
    ssl_certificate server.crt,
    ssl_certificate_key server.key;
    ssl_protocols TLSv1 TLSv1.1 TLSv1.2 TLSv1.3;
    location / {
        root /usr/share/nginx/html;
        index helloworld.html;
    }
}
```

Inoltre si aggiunge all'oggetto http:

```
ssl_session_cache shared:SSL:10m;
ssl_session_timeout 10m;
```

Java WebLogic Server

In WebLogic invece è molto più semplice in quanto la console ci permette tramite alcuni click di settare correttamente i parametri per instaurare il canale TLS.

1. Attivare SSL cliccando su 'Porta di ascolto SSL abilitata' sul Tab Generale del Server
2. Nel Tab Keystore inserire il percorso del keystore da usare, il suo tipo e il suo passphrase.
3. Nel Tab SSL inserire l'alias relativo alla chiave privata nel keystore e il suo passphrase.

Ciphersuites

Le ciphersuites utilizzate sono 2:

- TLS_AES_128_GCM_SHA256 - usata da Apache, Nginx e Tomcat
- TLS_AES_256_GCM_SHA384 - usata da Node e WebLogic

Considerazioni

Entrambe le ciphersuites sono ritenute sicure. NodeJs e WebLogic però hanno deciso di utilizzare una ciphersuites che utilizza una cifratura simmetrica più sicura che va però a discapito delle performance. Stessa cosa per l'algoritmo di hashing. Per quanto riguarda la versione del protocollo HTTP utilizzato, i primi 4 webserver si trovano allo stesso livello, infatti di default essi utilizzano HTTP 1.1. Per quanto riguarda WebLogic il protocollo HTTP utilizzato è di default HTTP2.

2.3.3 Instaurazione del canale sicuro TLS 1-way tra client e webserver con certificato server rilasciato da una CA privata.

Le configurazioni dei webserver devono essere modificate affinche si utilizzi il certificato rilasciato dalla CA privata istanziata tramite la configurazione vista in precedenza. Inoltre è necessario anche importare il certificato della root CA privata nella trust area dei vari browser.

Generazione richiesta di Certificazione in OpenSSL

Per richiedere un certificato alla CA, abbiamo bisogno di generare una CSR. Si utilizza un file di configurazione per settare l'estensione SAN:

```
[dn]
countryName          = "IT"
```

```
organizationName    = "Example"
commonName          = "www.helloworld.com"

[req]
distinguished_name  = dn
prompt              = no
req_extensions      = req_ext

[req_ext]
subjectAltName      = DNS:www.helloworld.com
```

Questo file specifica le estensioni x.509 da aggiungere in fase di creazione del certificato. Se non aggiungiamo l'estensione subjectAlternativeName al certificato il browser reputa il certificato non valido in quanto il campo CommonName non viene più utilizzato per identificare l'host.
Adesso con openssl è possibile generare la csr:

```
#openssl req -config reqconfig.cnf -reqexts req_ext \
# -new -key server.key -out server.csr
```

Con il file csr è possibile fare richiesta a qualsiasi CA, in questo caso utilizziamo la CA privata istanziata da noi, che ci rilascia un file server.crt. A questo punto i webserver devono specificare il certificato ottenuto come certificato da inviare durante l'handshake TLS.

Aggiungere la catena di certificati per raggiungere il Top-Level CA

Il certificato da solo non basta, i client hanno nella trust area il certificato della root CA e il certificato del server è firmato dalla sub CA. Quindi è necessario inserire nel certificato inviato dal server la catena di certificati per arrivare fino alla root CA:

```
#cat sub-ca.crt server.crt > chain.crt
```

E nel caso dei webserver che utilizzano un keystore:

```
#keytool -import -alias server \
# -file chain.crt -keystore .keystore
```

Apache HTTP Server

Specifichiamo la catena di certificati nel virtualhost del file httpd.conf:

```
SSLCertificateChainFile "/usr/local/apache2/keys/chain.crt"
```

Tomcat

Nel caso di Tomcat, non è necessario modificare il file di configurazione perchè la catena è già stata aggiunta al keystore con il comando keytool.

NodeJs

Nel file app.js il campo cert delle opzioni passate a createServer() va cambiato con la catena:

```
cert: fs.readFileSync(
    path.join(__dirname, 'keys', 'chain.crt')
),
```

Nginx

In Nginx è necessario modificare il campo ssl_certificate nell'oggetto server con la catena:

```
ssl_certificate chain.crt
```

Java WebLogic Server

WebLogic si comporta in modo simile a Tomcat in quanto utilizza i keystore per inviare i certificati. Quindi non è necessario aggiungere o modificare ulteriormente la configurazione una volta importati i certificati nel keystore. L'unica cosa che cambia è la generazione della CSR a partire da una coppia di chiavi pubblica/privata salvata nel keystore:

```
#keytool -certreq -alias server_key \
# -ext SAN=DNS:www.helloworld.com \
# -file server.csr -keystore .keystore
```

Binding DNS

Ogni browser quando riceve un certificato, controlla che il campo dell'estensione x.509 SubjectAlternativeName matchi il nome di dominio del webserver per validare il certificato. Per questo motivo è necessario creare all'interno del server DNS un nuovo Resource Record che bindi l'indirizzo IP del serverweb al nome di dominio inserito nel campo SubjectAlternativeName. Possiamo semplicemente aggiungere al file hosts del client la seguente direttiva:

```
127.0.0.1   www.helloworld.com
```

Da ora in poi le richieste dei browser ai serverweb dovranno usare come nome di dominio www.helloworld.com. E i certificati ricevuti saranno ritenuti validi.

2.3.4 Instaurazione del canale sicuro TLS 2-way tra client e webserver con certificati client e server rilasciati da una CA privata.

Nel TLS 2-way il server si aspetta che anche il client invii un certificato. La maggior parte delle attività di questa sezione sarà quindi svolta client-side in quanto dovremmo generare dei certificati client validi accettati dai webserver.

Binding chiave a certificato

Supponiamo che il browser del client sia installato su una macchina Windows. Dobbiamo allora trovare un modo per importare nella sua identity area le chiavi generate e i certificati rilasciati. Per quanto riguarda la generazione della coppia di chiavi e del certificato, questa procedura rimane la stessa, possiamo usare openssl. A questo punto è necessario linkare il certificato e la chiave privata in un file .pfx da importare nella identity area con:

```
#openssl pkcs12 -export -out client.pfx \
# -inkey client.key -in client.crt
```

Il file client.pfx è importabile nella identity area come se fosse un normale certificato. A questo punto tutti i browser installati possono fare riferimento a questo certificato e alla sua relativa chiave tutte le volte che viene richiesta l'autenticazione lato client.

Apache HTTP Server

In Apache l'autenticazione lato client può essere richiesta su alcune pagine html appartenenti ad una directory, andando a modificare il file http.conf nell'oggetto Directory. Si richiede la verifica del certificato lato client controllando fino ad un massimo di 2 certificati concatenati e si assegna il valore del CommonName allo Username del client:

```
DocumentRoot "/usr/local/apache2/htdocs"
<Directory "/usr/local/apache2/htdocs">
    Options Indexes FollowSymLinks
    AllowOverride None
    Require all granted
    SSLOptions +StdEnvVars
    SSLVerifyClient require
    SSLVerifyDepth 2
    SSLUserName SSL_CLIENT_S_DN_CN
</Directory>
```

Inoltre è necessario anche modificare l'oggetto VirtualHost per inserire il certificato della CA. Il Server Web controllerà che il certificato ricevuto dal client sia firmato con la chiave pubblica relativa al certificato della CA specificata:

```
SSLCACertificateFile "/usr/local/apache2/keys/root-ca.crt"
```

Problema con Apache

Apache di default utilizza TLSv1.3 come protocollo e richiede che, nel caso in cui sia stata abilitata l'autenticazione client, il browser effettui la Post Handshake Authentication.
La Post Handshake Authentication non è supportata nè da Chrome nè da Edge, in quanto pone dei problemi di sicurezza anche secondo l'AGID (Si veda la sezione relativa allo stato dell'Arte). In Firefox può essere abilitata andando nella pagina about:config selezionando true nel campo security.tls.enable_post_handshake_auth.
Nel caso si utilizzasse i browser Edge e Chrome è necessario specificare il downgrade di protocollo alla versione TLSv1.2 aggiungendo nell'oggetto VirtualHost la seguente direttiva:

```
SSLProtocol -all +TLSv1.2
```

Tomcat

Creiamo un truststore con il certificato della CA e specifichiamolo nel Connettore all'interno del file server.xml:

```
#keytool -import -file root-ca.crt \
# -alias CA -keystore .truststore
```

Il Connettore dovrà aggiungere gli attributi per usare il trustore creato nell'oggetto SSLHostConfig:

```
certificateVerification="required"
truststoreFile="/root/.truststore"
truststorePassword="passphrase"
truststoreType="PKCS12"
```

NodeJs

Aggiungiamo dei nuovi attributi alle opzioni passate al metodo createServer():

```
requestCert: true,
rejectUnauthorized: true,
ca: fs.readFileSync(
    path.join(__dirname, 'keys', 'root-ca.crt')
)
```

Nginx

Aggiungiamo le due direttive per la verifica del client nell'oggetto Server:

```
ssl_verify_client on;
ss_client_certificate root-ca.crt;
```

Java WebLogic Server

Come su Tomcat, è necessario creare un truststore con il certificato del root CA. Dopo sarà necessario accedere alla console di WebLogic:

1. Nel Tab generale abilitare la 2-way Authentication selezionando nel menù a tendina di "Funzionamento certificati client Two Way"la voce "Certificati client richiesti e applicati".

2. Nel Tab Keystore, sotto sicurezza, inserire il percorso del truststore personalizzato, il suo tipo e la sua passphrase.

2.4 Multiplexing con Proxy TLS

A livello Enterprise spesso vediamo l'utilizzo di un dispositivo che si interpone nella comunicazione tra browser e server Web. Viene chiamato Proxy ed è utilizzato per disaccoppiare tra loro il browser e il Server Web per motivi di efficienza e sicurezza.
Lo use case più comune consiste nell'avere un pool di Server Web lato back-end che verranno interrogati ciclicamente attraverso degli algoritmi di bilanciamento del carico per prevenire un overload delle risorse ed impedire che le singole istanze non siano più disponibili.
Nel nostro caso il Proxy interrompe il canale TLS, in questo modo i Server Web non si devono preoccupare di assegnare delle risorse alle operazioni di cifratura e di verifica dei messaggi ricevuti, in quanto esse sono delegate al Proxy.
Anche in questo caso presento due livelli:

1. Instaurazione di un canale sicuro TLS 2-way tra client e reverse proxy con certificati rilasciati da una CA pubblica e comunicazione in chiaro tra reverse proxy e webserver.

2. Instaurazione di un canale sicuro TLS 2-way tra client e reverse proxy con certificati rilasciati da una CA pubblica e instaurazione di un canale sicuro TLS 2-way tra reverse proxy e webserver con certificati rilasciati da una CA privata.

Le istanze dei Proxy sono: Apache HTTP Server e Nginx, visto che possono operare anche in modalità proxy.

2.4.1 Architettura e separazione dei domini

Supponiamo che l'IP dell'interfaccia esterna del terminatore SSL venga restituito dopo il lookup al nome "www.helloworld.com" appartenente al dominio "helloworld.com". L'interfaccia interna appartiene al dominio interno "privatenet.com" e le è assegnato il nome "sslterm.privatenet.com".
Oltre al terminatore SSL altri server sono presenti all'interno del dominio "privtenet.com":

- Il ServerWeb ha interfaccia con nome "webserv.privatenet.com"
- La CA interna ha interfaccia con nome "privca.privatenet.com"

Le impostazioni di dominio sono settate modificando il file /etc/hosts dei vari nodi.

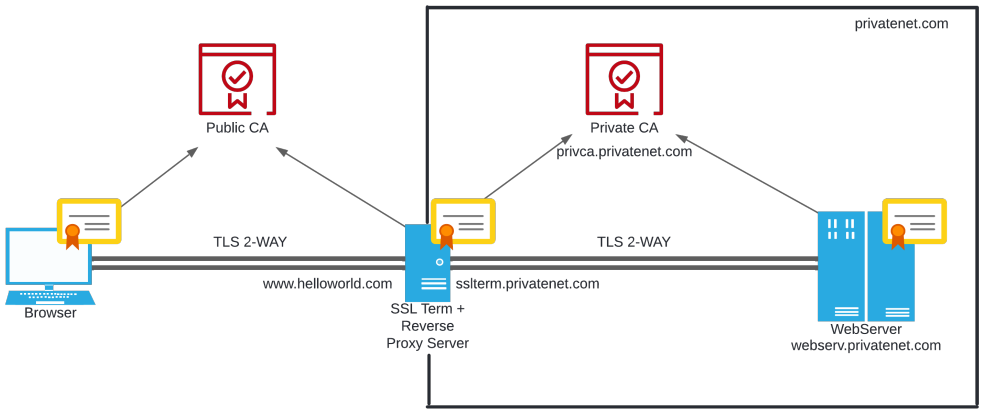

Figura 3: Rappresentazione ad alto livello dell'architettura da realizzare

2.4.2 TLS 2-way tra Client e Proxy (primo livello)

Sia nel caso del Proxy Apache che nel caso del Proxy Nginx il terminatore TLS deve instaurare una connessione TLS con il browser. Per fare questo è necessario creare con openssl una coppia di chiavi pubblica/privata, il relativo certificato e il chain file creato appendendo il certificato della CA. Inseriamo i file in una cartella, ad esempio in keys/extkeys. All'interno della cartella troviamo quindi: www.helloworld.com.key, www.helloworld.com.crt, chain.cert e root-ca.crt.

Nginx

Aggiungere le direttive che vengono utilizzate per impostare il timeout delle sessioni TLS in cache a 10 minuti nel file nginx.conf:

```
ssl_session_cache   shared:SSL:10m;
ssl_session_timeout 10m;
```

Specifichiamo la porta di ascolto, il nome di dominio del proxy e il keepalive delle sessioni TLS nell'oggetto Server. Settiamo anche le chiavi e i certificati per il processo di autenticazione mutuale. Si specificano infine i protocolli e si dice al proxy di verificare il certificato del client:

```
listen                  443 ssl;
server_name             www.helloworld.com;
keepalive_timeout       70;

ssl_certificate         keys/extkeys/chain.cert;
ssl_certificate_key     keys/extkeys/www.helloworld.com.key;
ssl_client_certificate  keys/extkeys/root-ca.crt;
ssl_protocols           TLSv1 TLSv1.1 TLSv1.2 TLSv1.3;
ssl_verify_client       on;
```

Adesso dobbiamo specificare le direttive che trasformano nginx in un reverse proxy. Esse vengono aggiunte alla direttiva location dentro l'oggetto server precedente.
Le prime 4 direttive specificano gli header da settare alla richiesta HTTP fatta dal proxy inviata al webserver. Esse servono per far conoscere al webserver le informazioni del browser (in quanto non è in contatto diretto). La quinta permette di inoltrare la richiesta al webserver specificando l'URL al quale risponde. Infine si disabilita il redirect in quanto non vogliamo delegare la richiesta al browser:

```
proxy_set_header        Host $host:80;
proxy_set_header        X-Real-IP $remote-addr;
proxy_set_header        X-Forwarded-For $proxy_add_x_forwarded_for;
proxy_set_header        X-Forwarded-Proto $scheme;
proxy_pass              http://webserv.privatenet.com/;
proxy_redirect          off;
```

Apache HTTP Server

Importare con LoadModule il modulo mod_proxy che permette la configurazione di Apache come Reverse Proxy.
Le prime 3 direttive da inserire nel VirtualHost vengono utilizzate per specificare il nome del server, abilitare il protocollo SSL e abilitare il Proxy:

```
ServerName www.helloworld.com
SSLEngine on
SSLProxyEngine on
```

Aggiungiamo le direttive relative al primo canale, quello cifrato tra browser e proxy, abilitando l'autenticazione mutuale:

```
SSLCertificateFile "www.helloworld.com.crt"
SSLCertificateKeyFile "www.helloworld.com.key"
SSLCertificateChainFile "chain.crt"
SSLCACertificateFile "root-ca.crt"
SSLProtocol -all +TLSv1.2
SSLOptions +StdEnvVars
SSLVerifyClient require
SSLVerifyDepth 2
SSLUsername SSL_CLIENT_S_DN_CN
```

E infine aggiungiamo la direttiva che permette di fare forwarding al server backend, che nel nostro caso è "webserv.privatenet.com":

```
ProxyPass / http://webserv.privatenet.com/
```

2.4.3 TLS 2-way tra Proxy e WebServer (secondo livello)

A livello Enterprise, molte volte, il traffico intranet deve essere protetto tanto quanto quello esterno. Per questo molte policy interne all'azienda decidono di applicare la cifratura anche ai canali instaurati internamente.
I passi da eseguire sono gli stessi della generazione di chiavi e certificati per TLS 2-way tra browser e proxy ma ovviamente in questo caso sia il proxy che il webserver devono farsi rilasciare i relativi certificati dalla CA privata "privca.privatenet.com". Chiamiamo server.key la chiave privata e server.crt il certificato del Webserver generati.
All'interno del Proxy creiamo una cartella che fa riferimento alle chiavi utilizzate per la comunicazione intranet chiamata intkeys:

```
#mkdir intkeys
#mv sslterm.privatenet.com.key sslterm.privatenet.com.crt \
# root-ca.crt > ./intkeys/
```

Generando di conseguenza il file chain.cert.

Nginx

All'interno dell'header Host dobbiamo specificare la porta 443 e nella direttiva che identifica l'URL a cui il webserver risponde dobbiamo modificare il protocollo in https. Poi dobbiamo aggiungere le direttive relative ai file utilizzati dall'autenticazione mutuale. Infine abilitare la verifica dei certificati, il riuso delle sessioni già instaurate e specificare i protocolli:

```
proxy_set_header                Host $host:443;
proxy_pass                      https://webserv.privatenet.com/;
proxy_ssl_certificate           keys/intkeys/chain.cert;
proxy_ssl_certificate_key       keys/intkeys/\
                                    sslterm.privatenet.com.key;
proxy_ssl_trusted_certificate   keys/intkeys/root-ca.crt;
proxy_ssl_verify_depth          2;
proxy_ssl_session_reuse         on;
proxy_ssl_protocols             TLSv1 TLSv1.1 TLSv1.2;
```

Notare che il file root-ca.crt in extkeys non è lo stesso del file root-ca.crt in intkeys, in quanto uno è il certificato della CA pubblica e l'altro è il certificato della CA privata.

Apache HTTP Server

In apache, dobbiamo modificare la direttiva ProxyPass per indicare https come protocollo e aggiungere le direttive relative al canale sicuro instaurato tra proxy e webserver. Le direttive da aggiungere sono quelle relative al certificato della CA privata (per validare i certificati inviati dai webserver), all'abilitazione della verifica dei certificati inviati dai webserver, alla specifica della versione del protocollo TLS e infine all'individuazione dei file dove le chiavi e i certificati risiedono:

```
SSLProxyCACertificateFile "keys/intkeys/root-ca.crt"
SSLProxyVerify require
SSLProxyVerifyDepth 2
SSLProxyMachineCertificateChainFile "keys/intkeys/chain.crt"
SSLProxyMachineCertificateFile "keys/intkeys/\
                                sslterm.privatenet.com.crt.key"
SSLProxyProtocol -all +TLSv1.2
ProxyPass / https://webserv.privatenet.com/
```

In Apache non esiste una direttiva tipo "SSLProxyMachineKeyFile" che permette di specificare la chiave privata da usare per la comunicazione. Per questo motivo è necessario appendere all'interno del file del certificato "sslterm.privatenet.com.crt.key" la chiave privata del proxy server.
All'interno di Nginx esistono alcune direttive che permettono di settare gli header relativi alla richiesta che il proxy server invia ai server backend. In Apache questa direttiva non esiste finchè non si importa il modulo mod_proxy_http che aggiunge di default gli header: X-Forwarded-Host, X-Real-IP, X-Forwarded-For, X-Forwarded-Proto e X-Forwarded-Server.

Capitolo 3

Autenticazione

Grazie al protocollo TLS, abbiamo instaurato un canale sicuro per la comunicazione tra client e server sia tra browser e proxy che tra proxy e backend server. Tuttavia, la sicurezza del canale di comunicazione non è sufficiente; è necessario aggiungere un ulteriore livello di sicurezza tramite l'Autenticazione degli utenti.
Nel capitolo sullo stato dell'arte, abbiamo già discusso i metodi di Autenticazione offerti dal protocollo HTTP: Basic e Digest Authentication. In questo capitolo, implementeremo il metodo Basic in tutti i Server Web già trattati, esamineremo un nuovo metodo chiamato FORM Authentication e, infine, presenteremo un caso d'uso specifico dell'Autenticazione tramite Identity Asserter Esterno in WebLogic Server e l'integrazione del Single Sign-On (SSO).
E' interessante capire come le uniche configurazioni da modificare sono quelle dei webserver di backend, questo perchè il reverse proxy è già stato configurato correttamente ed esso si limita a fare forwarding delle richieste e a restituire le risposte dei webserver.

3.1 Implementazione della Basic Authentication

Apache HTTP Webserver

In Apache abbiamo bisogno di creare con htpasswd il file delle password degli utenti. Htpasswd viene utilizzato per creare e aggiornare i file usati per memorizzare nomi utente e password per l'autenticazione di base degli utenti HTTP, esso hasha in automatico le password degli utenti:

```
htpasswd -c /usr/local/apache2/passwd/passwords webuser
```

L'utente webuser viene aggiunto al file con la password inserita interattivamente.
Adesso è possibile abilitare, tramite delle direttive, l'autenticazione sugli oggetti directory del file httpd.conf:

```
AuthType Basic
AuthName "File protetto"
AuthUserFile "/usr/local/apache2/passwd/passwords"
Require valid-user
Order allow,deny
Allow from all
```

Adesso ogni utente inserito all'interno del file passwords ha la possibilità di accedere alla pagina protetta. Order specifica l'approccio zero trust dove di default nessuno è trustato a meno che appartenente ad una whitelist (il file passwords).

Tomcat

Come ogni webserver basato su Java le applicazioni web sono in realtà dei file .war. Per questo Tomcat ha bisogno di specificare una struttura di default iniziale per poi creare il file da deployare:

```
basicauth/
    WEB-INF/
        web.xml
    jsp/
        index.jsp
```

La pagina index.jsp è la pagina protetta dalla Basic Authentication mentre web.xml serve per specificare gli attributi che l'applicazione deve avere (tra cui le pagine protette). Specifichiamo il fatto che i file dentro la cartella /jsp devono essere protetti da Basic Authentication in web.xml:

```
<security-constraint>
    <web-resource-collection>
        <web-resource-name>Risorsa Protetta</web-resource-name>
<url-pattern>/jsp/*</url-pattern>
        <http-method>GET</http-method>
    </web-resource-collection>
    <auth-constraint>
        <role-name>tomcat</role-name>
    </auth-constraint>
</security-constraint>
<login-config>
    <auth-method>BASIC</auth-method>
</login-config>
```

All'interno del tag auth-constraint è stato specificato che solo l'utente tomcat può accedere alle pagine sotto /jsp/.
A questo punto è necessario definire il file degli utenti. Ovvero tomcat-users.xml:

```
<tomcat-users>
    <role rolename="tomcat"/>
    <user username="tomcat" password="tomcat" roles="tomcat" />
</tomcat-users>
```

NodeJs

Per Node il discorso è diverso. Infatti è consigliato definire l'autenticazione come una funzione di middleware che viene utilizzata solo quando si tenta di accedere a delle risorse protette. Inoltre il file degli utenti deve essere creato,formattato in un certo modo ed essere acceduto con il modulo os. Per questo motivo, ho deciso di inserire lo username e la password hardcodate nel codice, e fare il confronto con le credenziali passate in input dall'utente. La funzione di middleware si presenta cosi':

```
function authentication(req, res, next) {
    const authHeader = req.header.authorization;
    if(!authHeader){
        let err = new Error("Non sei autenticato!");
        res.setHeader('WWW-Authenticate', 'Basic');
        err.status = 401;
        return next(err);
    }
    const auth = new Buffer.from(
        authHeader.split(' ')[1],
        'base64').toString().split(':');
    const user = auth[0];
    const pass = auth[1];
    if (user == 'admin' && pass == 'password')
        next();
    else {
        let err = new Error("Non sei autenticato!");
        res.setHeader('WWW-Authenticate', 'Basic');
        err.status = 401;
        return next(err);
    }
}
```

Nginx

Usare htpasswd per creare il file delle password ".htpasswd" come in apache, poi aggiungere l'oggetto location con valore uguale all'endpoint protetto nel file di configurazione:

```
location /protected {
    auth_basic "Area Protetta";
    auth_basic_user_file /etc/nginx/.htpasswd;
    index protected.html;
}
```

WebLogic Server

La struttura da creare nella command-line di Weblogic è molto simile a quella di Tomcat:

```
webpage/
    welcome.jsp
    WEB-INF/
        web.xml
        weblogic.xml
```

Visto che Weblogic mette a disposizione una console sulla porta 9002, possiamo utilizzarla per fare il deploy e per le successive operazioni.
Il file web.xml è lo stesso utilizzato da tomcat andando ovviamente a sostituire il nome della pagina protetta con welcome.jsp.
Il file weblogic.xml è un file che permette di mappare l'utente ad un gruppo specifico nel realm di weblogic:

```
<role-name>webuser</role-name>
<principal-name>myGroup</principal-name>
```

Creiamo il gruppo myGroup all'interno del realm. Creiamo l'utente webuser e assegnamolo al gruppo creato.

3.2 Implementazione della Form Authentication

La Form Authentication è una soluzione più evoluta rispetto alla Basic. Usa un modulo HTML dove l'utente inserisce nome utente e password. Dopo l'invio, il server verifica le credenziali e se valide, crea una sessione, inviando un cookie al browser per mantenere la sessione attiva. Le credenziali non vengono trasmesse ad ogni richiesta,

solo durante il login. Successivamente, il cookie gestisce la sessione. A differenza della Basic, permette una soluzione scalabile visto che tramite l'utilizzo dei token è possibile specificare le pagine protette alle quali l'utente ha accesso. Per generare i token ogni webserver fa utilizzo di algoritmi personalizzati.

Apache HTTP Server

Per prima cosa, visto che nella Form authentication non proteggiamo più le singole pagine ma proteggiamo gli endpoint raggiungibili, dobbiamo aggiungere una direttiva che permette di non avere conflitti tra i permessi:

```
Require all granted
```

Carichiamo i moduli:

```
auth_form_module
request_module
session_module
session_cookie_module
session_crypto_module
```

Aggiungiamo le direttive necessarie per proteggere un endpoint, ad esempio /protected, con la Form authentication:

```
Location /protected {
    AuthFormProvider file
    AuthUserFile "/usr/local/apache2/passwd/passwords"
    AuthType form
    AuthName "Area protetta"
    AuthFormLoginRequiredLocation \
        "https://www.helloworld.com/login.html"
    Session on
    SessionCookieName session path=/protected/
    Require valid-user
}
```

L'autenticazione avviene sempre utilizzando come Provider un file che contiene gli utenti e le loro password hashate. Si specifica l'utilizzo della Form Authentication e visto che si utilizzano le sessioni si aggiungono le direttive per la loro attivazione.
L'aggiunta della direttiva AuthFormLoginRequiredLocation permette di specificare la pagina di login sulla quale fare redirect nel caso un utente cerca di accedere ad una pagina protetta senza essersi autenticato.

Visto che la pagina di login contiene un form abbiamo bisogno di specificare il comportamento dopo che l'utente clicca su submit, ad esempio facendo una POST request all'endpoint /dologin:

```
Location /dologin {
    SetHandler form-login-handler
    AuthFormLoginRequiredLocation \
        "https://www.helloworld.com/login.html"
    AuthFormLoginSuccessLocation \
        "https://www.helloworld.com/protected/"
    AuthFormProvider file
    AuthUserFile "/usr/local/apache2/passwd/passwords"
    AuthType form
    AuthName "Area protetta"
    Session on
    SessionCookieName session path=/protected/
}
```

Infine, nel file HTML della pagina di login il form deve seguire lo standard di Apache specificando dei nomi precisi per gli attributi name dei tag input:

```
<input name="httpd_username"....>
<input name="httpd_password"....>
```

Tomcat

Aggiorniamo la struttura della web-app aggiungendo la pagina di login e la pagina di errore visualizzata nel caso in cui l'utente inserisca delle credenziali errate:

```
formauth/
    WEB-INF/
        web.xml
    index.jsp
    login.html
    login-failed.html
```

Il file HTML della pagina di login deve seguire lo standard di Tomcat utilizzando dei nomi appropriati per gli attributi e per l'action del form:

```
<form action="j_security_check"...>
    <input name="j_username"...>
    <input name="j_password"...>
```

Il file web.xml deve permettere la Form Authentication specificando FORM come auth_method e aggiungendo le pagine di login e di errore al tag form-login-config:

```
<security-constraint>
    <web-resource-collection>
        <web-resource-name>App protetta</web-resource-name>
        <url-pattern>/*</url-pattern>
        <http-method>GET</http-method>
        <http-method>POST</http-method>
    </web-resource-collection>
    <auth-constraint>
        <role-name>tomcat</role-name>
    </auth-constraint>
</security-constraint>
<login-config>
    <auth-method>FORM</auth-method>
    <form-login-config>
        <form-login-page>/login.html</form-login-page>
        <form-error-page>/login-failed.html</form-error-page>
    </form-login-config>
</login-config>
```

NodeJs

Creaiamo una cartella che contiene i file delle pagine login.html e index.html. Simuliamo un database aggiungendo un file data.js che contiene un array di utenti e esportiamo la struttura dati con module.exports che ci permette il suo utilizzo all'interno dello script principale.
L'endpoint /login che gestisce le richieste POST fatte dopo il click sul bottone di submit per l'autenticazione utente viene definito per confrontare le credenziali passate in input con le credenziali presenti nell'array che simula il database:

```
app.post('/login', async(req, res) => {
    try {
        let foundUser = users.find(
            (data) => req.body.user === data.user
        );
        if (foundUser) {
            let submittedPass = req.body.password;
            let storePass = foundUser.password;
            const passwordMatch =
```

```
                await bcrypt.compare(submittedPass, storePass);
            if (passwordMatch)
                res.send("Autenticato");
            else
                res.send("Email o password non valida");
        } else
            res.send("L'utente non esiste!");
    } catch {
        res.send("Internal Server Error");
    }
})
```

Questa forma di Form Authentication è rudimentale, infatti non consente l'invio di token per autenticare l'utente e questo fa si che per accedere alle pagine protette l'utente deve sempre passare per l'endpoint /login.

Nginx

Nginx, conosciuto per essere principalmente un Proxy, implementa la Form Authentication con due use-case differenti:

- Autenticazione basata su sottorichieste esterne
- Autenticazione con Token JWT

La prima, come dice il nome, non gestisce direttamente l'autenticazione ma fa riferimento a risorse esterne, ad esempio può essere utilizzato un Server AAA, un Server LDAP o un Identity Provider
La seconda invece gestisce l'autenticazione tramite l'utilizzo di token JWT gestiti da Nginx. Sfortunatamente questa seconda soluzione è supportata solo da Nginx Plus.
Per queste motivazioni la Form Authentication di Nginx può essere paragonata all'autenticazione con Identity Asserter esterno.

WebLogic Server

La struttura della web-app da deployare in weblogic definisce, come le web-app Java-based, i file JSP della pagina di login e della pagina di errore. Tutti i file definiti nella struttura possono essere copiati e incollati dai file di Tomcat, in quanto anche il formato del form definisce le stesse regole.

```
webpage/
    welcome.jsp
    login.jsp
```

```
errorpage.jsp
edit.jsp
WEB-INF/
    web.xml
    weblogic.xml
```

3.3 Configurazione di un Identity Asserter custom

Nei contesti Enterprise spesso è necessario delegare l'autenticazione a terze parti. Questo approccio risulta più scalabile ed elimina il problema legato al "single point of failure". In Weblogic server (e nella maggior parte dei webserver Enterprise) è possibile specificare un entità esterna che ha il compito di effettuare l'autenticazione degli utenti e di gestirne le identità. In Weblogic viene chiamato Identity Asserter.
Fino ad adesso abbiamo visto come Apache, Node e Tomcat (e anche Weblogic) permettono la gestione interna dell'autenticazione utilizzando come Provider un file degli utenti.
Con la prossima soluzione facciamo un'altro step, aggiungendo un server di autenticazione alla nostra architettura.
Utilizziamo come webserver un container Nginx che fa utilizzo di un server di autenticazione esterno per autenticare gli utenti.

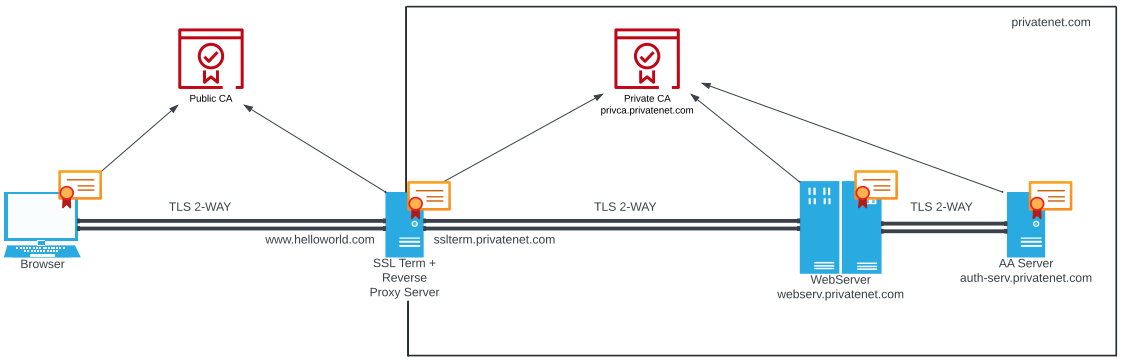

Figura 4: Rappresentazione ad alto livello dell'architettura da realizzare

3.3.1 Configurazione Server di Autenticazione

Il server di Autenticazione è un webserver Node che risponde sulla porta 3000 ed espone due API:

/dologin

Questa API controlla che lo username sia "admin" e la password sia "password", successivamente fa utilizzo di una funzione per generare un token. Una volta salvato in locale all'interno di un array lo invia settando l'Header Set-Cookie alla response HTTP. Infine fa redirect alla pagina protetta.

```
app.post('/dologin' (req,res) => {
    let submittedPass = req.body.password;
    let submittedUser = req.body.username;
    if (
        submittedPass == 'password' &&
        submittedUser == 'admin'
    ) {
        const token = genToken(10);
        res.cookie('authCookie', token,
            {maxAge: 900000, httpOnly: true}
        );
        cookiesArray.push(token);
        res.redirect('/protected/protected.html');
    }
});
```

/auth

Questa API controlla che il cookie inviato sia stato generato e che sia presente all'interno dell'array di cookie. La response status sarà 200 se il cookie esiste altrimenti ritorna 401. Questo è necessario per il webserver Nginx che si aspetta una di queste due response per l'autenticazione del client prima di rilasciare la pagina protetta.

```
app.get('/auth', (req, res) => {
    const cookieValue = req.cookies.authCookie;
    if (cookiesArray.includes(cookieValue))
        res.status(200).send();
    else
        res.status(401).send();
})
```

genToken()

La funzione genToken() crea una stringa di 10 caratteri casuali che rappresenta il token.

```
function genToken(length) {
    const characters = "ABCDEFGH......xyz0123456789";
    let result = '';
    for (let i = 0;i < length; i++)
        result += characters.charAt(
            Math.floor(Math.random()*characters.length)
        );
}
return result
```

3.3.2 Configurazione Web Server

Per specificare l'utilizzo di un Identity Asserter esterno in Nginx dobbiamo utilizzare la direttiva auth_request su tutte le pagine protette specificando l'endpoint sulla quale avverrà l'autenticazione esterna. L'accesso alle pagine protette nella directory /protected verrà seguito da un redirect sull'endpoint /auth che controllerà la presenza del cookie di autenticazione nella richiesta.

```
location /protected/ {
    auth_request /auth;
}
location = /auth {
    internal;
    proxy_pass                  https://auth-serv:3000/auth;
    proxy_pass_request_body     off;
    proxy_set_header            Content-Length "";
    proxy_set_header            X-Original-URI $request_uri;
}
location = /dologin {
    proxy_pass                  https:/auth-serv:3000/dologin;
    proxy_set_header            X-Original-URI $request_uri;
}
```

Quando Nginx va a fare forwarding della richiesta sull'endpoint /dologin dell'auth server è necessario tenere il body integro in quanto le credenziali dell'utente devono essere ricevute dal server di autenticazione. Questo non è altrettanto vero per l'endpoint /auth perchè il token verrà passato nell'header della richiesta HTTP.

login page

La action del form deve essere settata sull'endpoint /dologin, il quale genererà tutta la catena di autenticazione:

```
<form mehtod="POST" action="/dologin">
    Username: <input type="text" name="username"/>
    Password: <input type="password" name="password/>
    <input type="submit" name="login" value="Login"/>
</form>
```

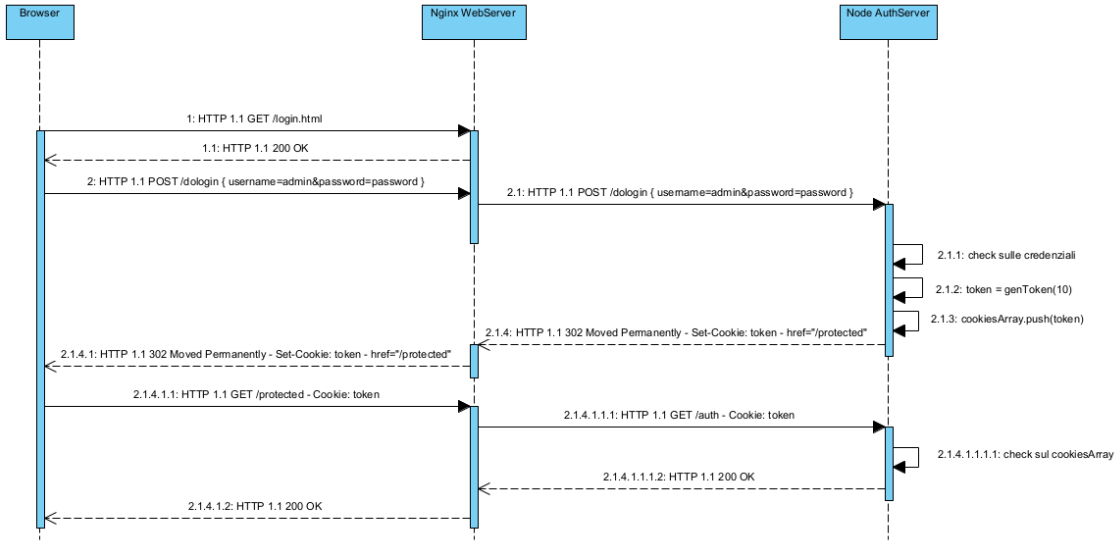

Figura 5: Rappresentazione del sequence diagram sullo use case di login

3.4 Single Sign On(SSO)

La configurazione di un Identity Asserter custom ci ha spinto verso un modo di autenticare che fa uso di Provider che sono nati con il solo compito di gestire le identità. Questo metodo di autenticare gli utenti può essere inteso come il 'prequel' dei moderni metodi di autenticazione che abbracciano l'approccio Single Sign On(SSO). Il prossimo step da compiere è quello di utilizzare l'architettura con Reverse Proxy e doppio canale TLS per configurare un ambiente di SSO. Questo significa che gli utenti, prima di poter accedere alle risorse, hanno bisogno di interfacciarsi con l'Identity Provider per l'autenticazione. Successivamente gli utenti possono accedere a qualsiasi risorsa rilasciata dai Service Provider appartenenti alla federazione.
L'architettura quindi prevede l'aggiunta delle seguenti entità: un Service Provider che implementa il protocollo SAML2, un Service Provider che implementa il protocollo OpenID Connect(OIDC), e un Identity Provider che gestisce le identità e autentica gli utenti.

Il reverse proxy simula anche un router perimetrale di una rete privata che contiene i Service Provider.
L'Identity Provider invece non appartiene alla rete privata ed è quindi accessibile direttamente dal browser dell'utente.

Istanze dei dispositivi

- Il Service Provider che implementerà il workflow SAML2.0 è un immagine WebLogic Server.
- Il Service Provider che implementerà il workflow OpenID Connect è un immagine Tomcat.
- L'Identity Provider è un immagine WSO2 Identity Server.

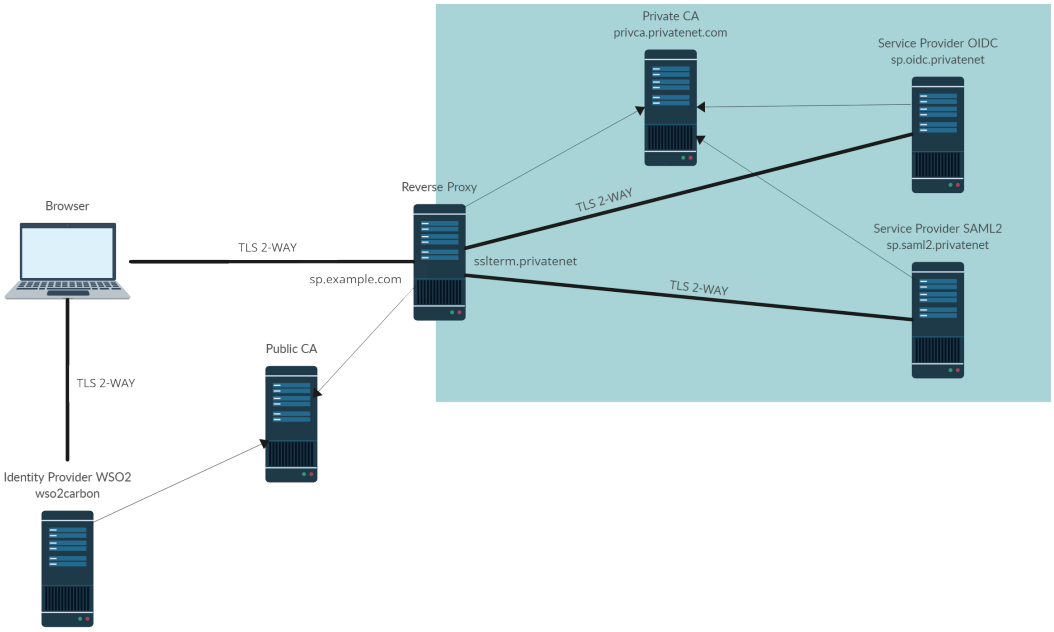

Figura 6: Rappresentazione dell'architettura da realizzare

La decisione di istanziare due Service Provider che implementano due protocolli per il SSO diversi è stata presa per capire meglio le differenze tra le due tecnologie e sapere quale delle due preferire in determinati contesti.
In particolare gli use-case sono:

- POST-Binding mutuale per SAML 2.0

- Authorization code flow per OIDC

Prima di mettere le mani sulla configurazione dei Service Provider è necessario configurare alcune impostazioni sul Identity Provider che andremo ad utilizzare.

3.4.1 Configurazione iniziale di WSO2

Hostname e TLS

Per creare un istanza di WSO2 (ad esempio in Docker) è importante modificare il file deployment.toml che contiene la configurazione di base del server WSO2. All'interno è possibile definire l'hostname, le credenziali per accedere alla console web, i binding ai database che contengono gli utenti (user stores) e i file di identitystore e di truststore che contengono le chiavi per la cifratura TLS.
Per evitare problemi sui lookup DNS setto l'hostname a wso2carbon.
Creo un keystore nuovo con una nuova coppia di chiavi pubblica/privata e setto nel certificato l'estensione SAN = wso2carbon (in questo modo il browser non da problemi di mismatch tra URL e certificato). Esporto il certificato dal keystore e lo importo nel truststore esistente. A questo punto modifico la configurazione del file deployment.toml ed indico che l'identitystore da usare è il nuovo keystore creato.
Il certificato adesso può essere inserito nella trust area del browser di Google Chrome dell 'utente.

Creazione utente

In WSO2 si utilizzano gli user stores per specificare i database utilizzati per archiviare le informazioni sugli utenti. Ad esempio è possibile specificare un DB Java-Based come user store ed è possibile connettersi con JDBC. Nel nostro esempio abbiamo bisogno di creare un solo utente e quindi lo creiamo direttamente tramite la console di WSO2.
Collegandosi all'URL https://wso2carbon:9443/console (dopo aver fatto il binding tra wso2carbon e 127.0.0.1 in /etc/hosts) è possibile accedere con le credenziali specificate nel file di deploy.
Spostandosi nella sezione User Management - Users è possibile fare click su Add User e fornire le informazioni di base del nuovo utente. Nel nostro esempio l'utente creato avrà username: "ssouser" e sarà l'utente che verrà riconosciuto sia dal Service Provider che dall'Identity Provider. Questa procedura, nei contesti Enterprise, viene automatizzata con il Provisioning, ovvero effettuando una sincronizzazione in rete tra SP e IdP.

3.4.2 Configurazione del POST-Binding mutuale SAML 2.0 su Weblogic

Configurazione webapp con IdP SAML 2.0

Il file di configurazione della webapp, ovvero web.xml contiene come auth-method la direttiva CLIENT-CERT, usata per specificare l'utilizzo di un Identity Asserter:

```
<security-constraints>
    <web-resource-collection>
        <web-resource-name>Protected pages</web-resource-name>
        <url-pattern>/admin/*</url-pattern>
        <http-method>GET</http-method>
    </web-resource-collection>
    <auth-constraint>
        <role-name>admin</role-name>
    </auth-constraint>
</security-constraints>
<login-config>
    <auth-method>CLIENT-CERT</auth-method>
</login-config>
<security-role>
    <role-name>admin</role-name>
</security-role>
```

Le pagine protette dal workflow di SAML 2.0 sono raggiungibili sotto l'endpoint /admin/ e solo il ruolo admin può accederci. Il file weblogic.xml deve bindare il role-name "admin" al principal "ssouser":

```
<security-role-assignment>
    <role-name>admin</role-name>
    <principal-name>ssouser</principal-name>
</security-role-assignment>
```

E' quindi necessario creare l'utente ssouser all'interno del realm di weblogic. La webapp quindi può essere deployata.

Trasformare WebLogic in un Service Provider SAML 2.0

Per prima cosa WebLogic deve essere in grado di poter instaurare connessioni cifrate con TLS, quindi dalla console di WebLogic è necessario abilitare la porta SSL.
Si specificano le informazioni del Service Provider, si va su Federation Services - Generale SAML2.0.

Il campo ID entità viene utilizzato per compilare il tag Issuer nella AuthnRequest, ovvero la richiesta che il SP fa al IdP per autenticare l'utente, ed è importante che lo stesso valore sia specificato anche sull'IdP. Ad esempio utilizziamo il nome "samlAP" come ID entità.
È necessario disabilitare il controllo dei destinatari. Se questo controllo è abilitato (impostato su "true"), il destinatario della richiesta SAML deve corrispondere all'URL della richiesta HTTP. Tuttavia, nella nostra architettura è presente un Reverse Proxy, il che significa che l'URL della richiesta sarà sempre diversa dal destinatario indicato nella richiesta.
Abilitare il servizio di Service Provider SAML2.0 dal suo tab.
Scaricare il file metadata.xml del Service Provider cliccando sul bottone publish metadata. Questo file contiene tutte le informazioni per effettuare il trust binding tra SP e IdP e dovrà quindi essere importato in WSO2.

Creazione dell'App di autenticazione in WSO2

Accedere nuovamente alla console di WSO2 e creare una nuova App Traditional Web-Page, selezionando il protocollo SAML2. Selezionare Upload e passare il file metadata.xml scaricato da WebLogic.
Nel tab Protocol dobbiamo specificare qual'è l'Assertion Consumer Service URL. Questo endpoint si occupa di ricevere le SAMLResponse generate dall'IdP sul Service Provider per controllare le informazioni dell'utente di modo che lo possa autenticare. Nel nostro caso quindi aggiungiamo https://sp.example.com/saml2/sp/acs/post. Selezionamolo come default URL.
A questo punto è importante specificare quale user attribute viene inserito all'interno della SAMLResponse per identificare l'utente. Di default WSO2 utilizza lo User-ID, ovvero una stringa che viene creata dinamicamente quando si crea un nuovo utente. Nel nostro caso vogliamo che l'utente venga identificato con il suo username, ovvero "ssouser" (altrimenti l'Authenticator di WebLogic non saprebbe quale utente mappare e restituirebbe Forbidden Error). Si aggiunge quindi il claim dal Tab User Attribute chiamato https://wso2.org/claims/username e selezioniamo nella sezione Subject il campo username al posto dello User-ID.
Infine dal Tab Info si scarica il file metadata.xml che contiene le informazioni per fare il trust binding alterno. Questa volta il file verrà utilizzato da Weblogic per conoscere quale Identity Asserter esterno chiamare.

Configurazione Identity Asserter SAML2 su WebLogic

Nella sezione realm di sicurezza si aggiunge il Provider SAML2IdentityAsserter. Spostarsi nel suo tab Gestione ed aggiungere un Web SSO idP partner. In questo caso si utilizza il file metadata.xml scaricato da WSO2.

Infine specificare le Redirect URI ovvero la URI sulla quale fare redirect una volta che gli utenti vengono autenticati. Nel nostro caso /appB/admin/services.jsp.

Configurazione dell'IdentityAuthenticator SAML2

Nel Tab Provider aggiungere un nuovo provider SAMLAuthenticator. Spostarsi nel suo tab Common e selezionare il Control Flag su Sufficient. Selezionare il Control Flag su Sufficient anche sul DefaultAuthenticator.
La direttiva SUFFICIENT, viene usata per specificare il fatto che è sufficiente l'autenticazione usando un solo Provider per autenticarsi al Service Provider.

Configurazione Reverse Proxy Nginx

Le direttive da aggiungere alla Location / in nginx.conf sono:

```
proxy_set_header Host $host:7002;
proxy_set_header X-Real-IP $remote_addr;
proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
proxy_set_header X-Forwarded-Proto https;
proxy_pass       https://sp.saml.privatenet:7002/;
proxy_redirect   https://sp.example.com:7002/ \
                            https://sp.example.com/;
```

La direttiva Proxy Pass utilizza un URL che identifica internamente il Service Provider (necessario quindi bindare sp.saml.privatenet all'IP del Service Provider WebLogic in /etc/hosts).
Anche nel backend side viene utilizzata la cifratura TLS, è necessario quindi aggiungere tutte le direttive per abilitare TLS e per specificare chiavi e certificati ottenuti dalla Private CA.
Il SP dopo aver ricevuto la SAMLResponse sull'endpoint /saml2/sp/acs/post fa redirect dell'utente a https://sp.example.com:7002/appB/admin/services.jsp ma questa pagina non viene riconosciuta dal TLS terminator in quanto ha come porta aperta solo la 443. Per questo motivo con la direttiva proxy_redirect si effettua il replace dell'URL e si utilizza la porta 443 (non è necessario specificarla).

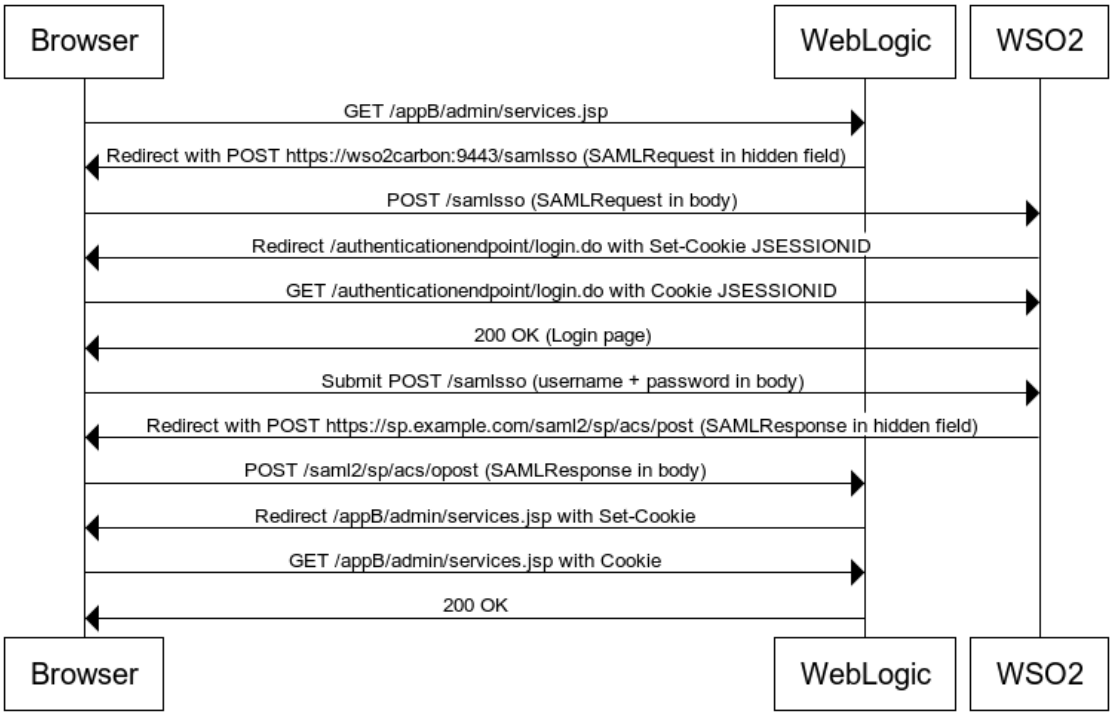

Figura 7: Rappresentazione del sequence diagram sullo use-case di accesso ad una risorsa protetta

3.4.3 Configurazione dell'Authorization code flow OIDC su Tomcat

Installazione OIDCAuthenticator in Tomcat

Tomcat include un certo numero di Authenticators per i meccanismi standard di autenticazione come HTTP Basic, HTTP Digest e HTTP Form. Questi Authenticators sono implementati attraverso delle "Valve" all'interno del file server.xml di Tomcat. Di default, però, non sono presenti degli Authenticator che permettono l'interfacciamento con un provider OpenID Connect per loggare gli utenti. Per questo motivo è stato creato un Authenticator, da terze parti, che estende l'autenticazione form-based e aggiunge l'implementazione dei flow OpenID Connect.
Il progetto github è presente al seguente indirizzo:
https://github.com/boylesoftware/tomcat-oidcauth.
Per installarlo è sufficiente inserire il jar nella cartella lib di $CATALINA_BASE di Tomcat.

Creazione dell'App di autenticazione in WSO2

Accedere alla console di WSO2 e creare una nuova Traditional Web-Page Application e selezionare OpenID Connect.
Nel campo Authorized redirect URLs inserire l'URL che si occuperà di mappare l'Access Token ricevuto dall'IDP con un utente locale presente nel file degli utenti.
Questa funzione in Tomcat è eseguita dall'endpoint /j_security_check quindi l'URL da inserire nel nostro caso è:
https://sp.example.com/sp/j_security_check. (Qui ho passato l'hostname del reverse proxy in quanto il SP non è accessibile direttamente).
Nel Tab Protocol sono presenti i campi: clientId e clientSecret. Questi campi verrano utilizzati dall'OIDCAuthenticator per generare una trust binding tra WSO2 e Tomcat. Nel Tab info sono presenti tutti gli endpoint necessari ai SP per eseguire il flow OIDC.

Configurazione dell'OIDCAuthenticator

Nel file server.xml di Tomcat si specifica una nuova Valve:

```
<Valve className=
    "org.bsworks.catalina.authenticator.\
        oidc.tomcat90.OpenIDConnectAuthenticator"
    providers="[
        {
            name: Wso2,
            issuer: https://wso2carbon:9443/oauth2/token,
            configUrl:
                 https://wso2carbon:9443/oauth2/\
                        token/.well-known/openid-configuration,
            clientId: fH4y7.....ioa,
            clientSecret: HWD0Z...cca
        }
    ]"
    usernameClaim="username" additionalScopes="profile" />
```

La Valve richiede la specifica obbligatoria di almeno un provider, nel nostro caso il provider è WSO2. Le informazioni richieste dalla specifica di un Provider sono recuperate dal Tab informazioni dell'App di autenticazione creata in precedenza su WSO2. E' necessario specificare i campi opzionali usernameClaim e additionalScopes in quanto WSO2 di default richiede la specifica da parte dei SP, per ottenere informazioni sullo username dell'utente da autenticare, di una richiesta contenente come Scope: openid e profile. Infatti il claim "username" è sottocategoria dello Scope

"profile". Di default lo scope openid viene inserito da Tomcat e non è necessaria una dichiarazione esplicita.

Configurazione utenti in Tomcat

Il file tomcat-users.xml di Tomcat consente di specificare ruoli e utenti. L'utente precedentemente creato su WSO2 è "ssouser" e l'Authenticator di Tomcat deve mappare l'identità dell'utente autenticato dall'IdP a questo utente.
Il file tomcat-users.xml può quindi definire in questo modo il ruolo ssouser:

```
<role rolename="ssouser"/>
<user username="ssouser" password="pass" roles="ssouser" />
```

Configurazione webapp con IdP OIDC

Il file di configurazione web.xml contiene:

```
<security-constraints>
    <web-resource-collection>
        <web-resource-name>Protected Pages</web-resource-name>
        <url-pattern>/admin/*</url-pattern>
        <http-method>GET</http-method>
    </web-resource-collection>
    <auth-constraint>
        <role-name>ssouser</role-name>
    </auth-constraint>
</security-constraints>
<login-config>
    <auth-method>FORM</auth-method>
    <form-login-config>
<form-login-page>/login.jsp</form-login-page>
<form-error-page>/login-failed.jsp</form-error-page>
</form-login-config>
</login-config>
<security-role>
    <role-name>ssouser</role-name>
</security-role>
```

Notare come questo file è molto diverso dal file web.xml di WebLogic. L'auth-method specificato è FORM e non CLIENT-CERT. Infatti tale keyword è proprietaria di WebLogic e non può essere utilizzata in Tomcat.
Si specifica la pagina di login e la relativa pagina di errore sulla quale reindirizzare

gli utenti a seguito di un errore nel flow di autenticazione.
All'interno della cartella /WEB-INF/lib/ si specificano le librerie utilizzate dalla pagina web in fase di rendering della pagina. La libreria JSTL permette di accedere alle request attribute che contengono i Bean utilizzati per accedere agli attributi degli utenti autenticati.
I file jar da importare di questa libreria sono presenti alla pagina:
https://tomcat.apache.org/taglibs/standard/
La pagina di login.jsp utilizza la libreria JSTL per accedere agli Endpoints definiti nella Valve precedente sottoforma di array.

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>
<c:redirect url="${
    requestScope['org.bsworks.oidc.authEndpoints'][0].url
}"/>
```

Quando l'utente accede alla pagina di login verrà semplicemente redirezionato all'endpoint di autenticazione di WSO2.
La pagina di errore configurata nel tag form-login-config di web.xml ha a disposizione un request attribute sotto "org.bsworks.oidc.error", il quale è un bean con gli attributi: code, description e infoPageURI. Nella pagina di errore visualizzo il codice di errore definito dall'RFC della specifica di OpenID Connect:

```
<c:set var = "code" value = "${
    requestScope['org.bsworks.oidc.error.code']
}"/>
<c:out value = "${code}"/>
```

Nella homepage dell'utente autenticato posso far visualizzare l'Issuer ID dell'OpenID-Connect Provider. Si accede al session attribute sotto 'org.bsworks.oidc.authorization' il quale espone l'attributo Issuer e lo si visualizza a schermo con "c:out".

Configurazione Reverse Proxy Nginx

Il reverse proxy deve poter identificare a quale dei due SP la richiesta, che proviene dal browser, deve essere inviata. Con la direttiva Location è possibile suddividere facilmente le richieste tramite delle espressioni regolari che matchano determinate parole chiave presenti nell'URL della richiesta.
Nel nostro caso sappiamo che le richieste che hanno nell'URL il percorso iniziale /appB/* o /saml2/* sono destinate al SP SAML2 (WebLogic) mentre le richieste che hanno nell'URL il percorso iniziale /sp/* sono destinate al SP OIDC (Tomcat).
La direttiva Location /sp/* si differenzia dalla direttiva Location /appB/* e /saml2/* solo dal server backend e quindi si specifica:

```
proxy_set_header Host $host:8443;
proxy_set_header X-Real-IP $remote_addr;
proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
proxy_set_header X-Forwarded-Proto https;
proxy_pass       https://sp.oidc.privatenet:8443/;
proxy_redirect   https://sp.example.com:8443/ \
                           https://sp.example.com/;
```

La porta specificata ed esposta da Tomcat è la 8443 e richede di collegarsi con HTTPS. La cifratura TLS anche in questo caso deve essere configurata all'interno del Connettore Tomcat.
La figura sottostante

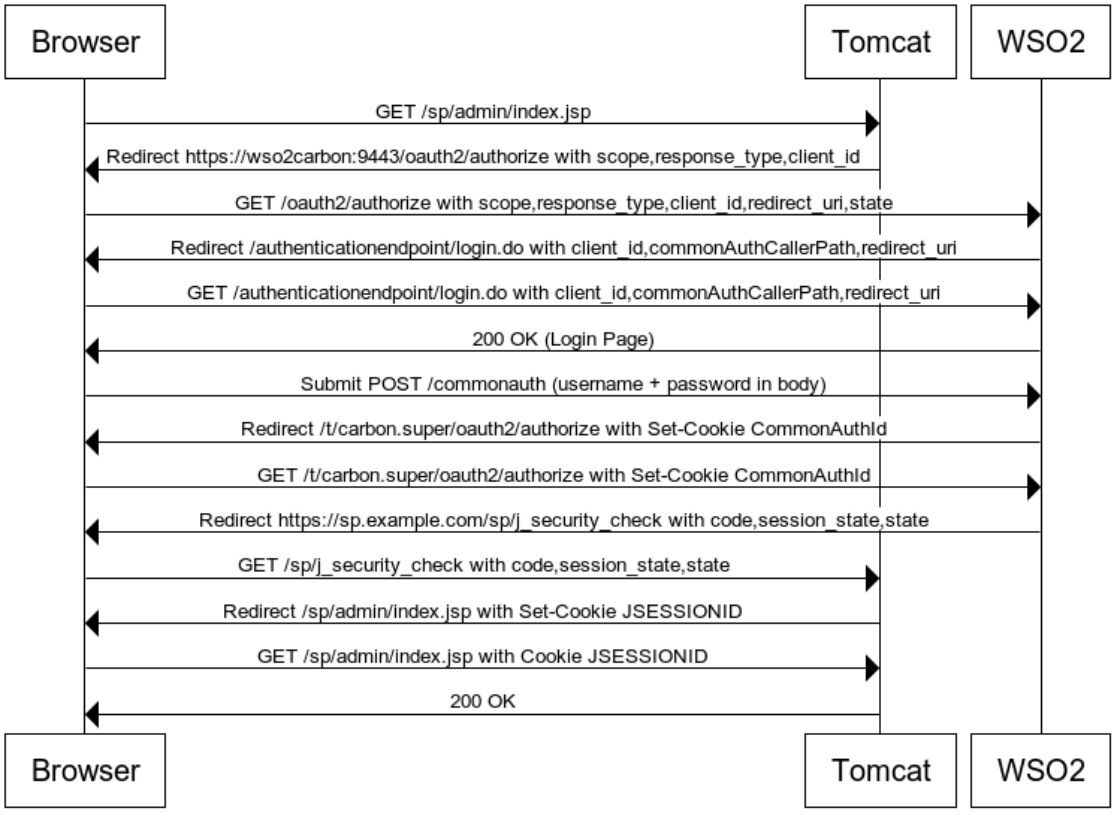

Figura 8: Rappresentazione del sequence diagram sullo use-case di accesso alle risorse protette

Capitolo 4

Autorizzazione

Mentre l'autenticazione si occupa di verificare l'identità di un utente, l'autorizzazione determina a quali risorse un utente autenticato può accedere e quali azioni può compiere su di esse. Per questo l'autorizzazione è considerata un pilastro fondamentale nella sicurezza dei sistemi informatici e delle applicazioni web.
Il processo di autorizzazione è essenziale per proteggere le risorse sensibili e garantire che solo gli utenti con i permessi appropriati possano eseguire operazioni specifiche.
Ogni sistema di Autorizzazione è composto principalmente da 4 entità: i soggetti che richiedono l'accesso alle risorse, gli oggetti a cui si richiede l'accesso come file, database o servizi, le azioni ovvero le operazioni che possono essere eseguite sulle risorse e le regole ovvero delle condizioni da soddisfare per concedere un azione di un soggetto su un oggetto.
L'autorizzazione viene gestita attraverso diversi modelli e ciascuno offre vari livelli di flessibilità e complessità, adattandosi a diverse esigenze e scenari di utilizzo:

- Access Control List (ACL): Una lista associata a una risorsa che specifica quali utenti o quali gruppi possono accedere a quella risorsa e con quali permessi.
- Role-Based Access Control (RBAC): Gli utenti sono assegnati a ruoli e i ruoli sono assegnati a permessi, semplificando la gestione dei permessi e permettendo alta scalabilità in contesti Enterprise medio/grandi.
- Attribute-Based Access Control (ABAC): Le decisioni di accesso sono baste su attributi dell'utente, dell'ambiente e delle risorse, offrendo maggiore flessibilità rispetto a RBAC.

In questo capitolo, l'obiettivo è stabilire una baseline comune per tutti i server web analizzati. Questa baseline consiste nella creazione di tre gruppi con un totale di cinque utenti distribuiti come segue:

- Nel gruppo admin troviamo l'utente Lorenzo.

- Nel gruppo sales troviamo gli utenti Sara e Filippo.
- Nel gruppo devs troviamo gli utenti Antonio e Andrea.

Gli utenti del gruppo admin hanno accesso sia alle risorse riservate al gruppo sales che a quelle del gruppo devs.
Una volta raggiunta questa baseline, si procederà con l'integrazione di policy più granulari per ciascun server web, sfruttando le specifiche funzionalità offerte dalle rispettive tecnologie.

4.1 Autorizzazione in Apache HTTP Server

Apache supporta il modello Access Control List(ACL), infatti permette la loro gestione, specificando controlli basati su caratteristiche del dispostivo (ad esempio l'IP o il domain-name) oppure basate sull'utente autenticato controllando gli eventuali gruppi a cui appartiene.
E' possibile anche configurare ABAC tramite moduli come mod_authz_core, che valutano vari attributi per determinare l'accesso (ad esempio variabili di ambiente settate).
Infine non supporta RBAC nativamente, ma può essere configurato per simularlo tramite estensioni o moduli di terze parti.

4.1.1 Baseline con Groupfile

Creazione groupfile e directory tree

La baseline definita è facilmente implementabile su Apache tramite la configurazione di un file chiamato groupfile, il quale definisce i gruppi di sistema e gli utenti inclusi in ciascun gruppo. È essenziale che questo file segua una formattazione standard per garantire la corretta gestione degli accessi.

```
admin: lorenzo
sales: filippo sara
devs: antonio andrea
```

Ovviamente ogni utente deve contenere una entry nel file delle password. Tale file può essere costruito tramite il tool htpasswd come abbiamo già visto precedentemente.
Ogni gruppo ha la sua pagina protetta all'interno della cartella protected/:

```
protected/
    admin.html
    sales.html
```

```
    devs.html
login.html
index.html
```

Configurazione permessi

Apriamo il file httpd.conf e proteggiamo con FORM Authentication la directory /protected. Aggiungiamo la direttiva che specifica il percorso del groupfile da utilizzare per il controllo dei gruppi:

```
<Directory "/protected">
    Options Indexes FollowSymLinks
    AllowOverride None
    Require all granted
    AuthUserFile "/usr/local/apache2/passwd/passwords"
    AuthGroupFile "/usr/local/apache2/passwd/groupfile"
    AuthFormLoginRequiredLocation
        "https://www.helloworld.com/login.html?req=%{REQUEST_URI}"
    AuthFormProvider file
    AuthType form
    AuthName "Area ristretta"
    Session on
</Directory>
```

All'interno dell'URL specificato dalla direttiva AuthFormLoginRequiredLocation abbiamo aggiunto un parametro che specifica a quale pagina l'utente non autenticato ha cercato di accedere, così che la pagina di login sappia dove redirezionare l'utente una volta finito il processo di autenticazione.
Configuriamo il controllo degli accessi sulle risorse:

```
<Location "/protected/sales.html">
    SessionCookieName session path=/protected/
    Require group sales admin
</location>
<Location "/protected/admin.html">
    SessionCookieName session path=/protected/
    Require group admin
</location>
<Location "/protected/devs.html">
    SessionCookieName session path=/protected/
    Require group devs admin
</location>
```

La direttiva Require permette di specificare il gruppo al quale l'utente deve appartenere affinchè possa accedere alla risorsa specificata. Notare come la direttiva permette l'accesso alle risorse di ogni gruppo da parte degli utenti admin.
La direttiva SessionCookieName specifica il nome del cookie di sessione da inviare al client con l'aggiunta dell'attributo path che specifica dove tale cookie può essere inviato.
Infine si configura il comportamento dell'endpoint /dologin.html, ovvero l'endpoint raggiunto dopo aver cliccato sul bottone di submit nella login page:

```
<Location "/dologin.html">
    SetHandler form-login-handler
    Session On
    SessionCookieName session path=/
    SetEnvIf Referer ^.req=(.*)&?$ req=$1
    AutUserFile "/usr/local/apache2/passwd/passwords"
    AuthGroupFile "/usr/local/apache2/passwd/groupfile"
    AuthFormLoginRequiredLocation
        "https://www.helloworld.com/login.html"
    AuthFormProvider file
    AuthType form
    AuthName "Area Ristretta"
</Location>
```

La parte che esegue il login è l'handler definito con la direttiva SetHandler ovvero il form-login-handler. Viene settata la variabile req con valore uguale al valore del header Referer della richiesta HTTP.

Pagina di login

La pagina di login deve essere adattata alla configurazione specificata nel file httpd.conf. Si aggiunge quindi un input nascosto all'interno della pagina che verrà valorizzato con il percorso corrente sulla quale l'utente sta navigando:

```
<input id="loc" type="hidden" name="httpd_location" value="" />
```

Per settare il valore di loc si usa del codice javascript da aggiungere alla pagina di login che prende il valore req dalla URL e lo inserisce nell'attributo value:

```
let loc = document.getElementById("loc");
const queryString = window.location.search;
const urlParams = new URLSearchParams(queryString);
loc.setAttribute("value", urlParams.get('req'));
```

4.1.2 Integrazione con utenti definiti in altri Provider

Fino ad ora abbiamo implementato le policy di autorizzazione utilizzando utenti definiti in risorse statiche come il file delle password e il groupfile. Tuttavia, questo approccio non è scalabile in realtà medio/grandi dove le utenze sono dinamiche e soggette a frequenti modifiche. In contesti di questo tipo, è preferibile gestire gli utenti utilizzando strutture dati più adatte e flessibili.
Apache mette a disposizione 3 moduli per definire policy di autorizzazione compatibili direttamente con queste architetture:

- mod_authnz_ldap: questo modulo permette operazioni di autenticazione e autorizzazione che estendono le funzionalità del modulo mod_auth_basic per l'interfacciamento con una directory ldap (anche con Active Directory).
- mod_authz_dbd: questo modulo permette il login e l'identificazione del gruppo di appartenenza di un utente attraverso l'interrogazione di un DB SQL.
- mod_authz_dbm: questo modulo permette l'identificazione del gruppo di appartenenza di un utente attraverso l'accesso ad un file DBM.

Configurazione di Policy di autorizzazione per utenti definiti in una directory LDAP

Durante la fase di autorizzazione, il modulo mod_authnz_ldap prova a determinare se l'utente ha i permessi necessari per accedere alla risorsa. Per determinarlo, il modulo, effettua dei controlli attraverso delle operazioni di confronto sul server LDAP configurato.
Per aggiungere un Server LDAP con cui Apache si interfaccerà si utilizza la direttiva AuthLDAPURL valorizzata con un URL che segue lo standard definito dall'RFC 2255:

```
AuthLDAPURL "ldap://ldap.helloworld.com"
```

Supponiamo che siano presenti le seguenti entry nel server LDAP:

```
dn: cn=Admin, o=Example
objectClass: groupOfUniqueNames
uniqueMember: cn=Lorenzo Fallani, o=Example

dn: cn=Sales, o=Example
objectClass: groupOfUniqueNames
uniqueMember: cn=Sara Milli, o=Example
uniqueMember: cn=Filippo Gustosi, o=Example
```

```
dn: cn=Devs, o=Example
objectClass: groupOfUniqueNames
uniqueMember: cn=Antonio Birbanti, o=Example
uniqueMember: cn=Andrea Bibbieni, o=Example
```

Il controllo degli accessi può essere quindi effettuato sulla base di: utenti, distinguished-names, gruppi e attributi utilizzati all'interno del server LDAP, grazie alla direttiva Require combinata con il modulo appena importato.
Proteggiamo, ad esempio, la pagina sales.html come abbiamo fatto precedentemente ma questa volta utilizzando l'interfacciamento con il server LDAP. Si modifica la Location in questo modo:

```
<Location "/protected/sales.html">
    SessionCookieName session path=/protected/
    <RequireAny>
        Require ldap-group cn=Administrators, o=Example
        Require ldap-group cn=Sales, o=Example
    </RequireAny>
</Location>
```

L'utilizzo della direttiva RequireAny viene usata per specificare che è sufficiente appartenere ad uno dei due gruppi per ottenere l'accesso.

4.2 Autorizzazione in Tomcat

Tomcat supporta il modello Access Control List(ACL), infatti tramite la configurazione di web.xml e tomcat-users.xml si riesce a specificare quali utenti possono accedere a determinate risorse.
Supporta nativamente il modello RBAC, consentendo la definizione di ruoli e l'assegnazione di permessi a questi ruoli sempre attraverso il file tomcat-users.xml.
Infine non supporta ABAC nativamente, ma può essere esteso per supportarlo tramite l'uso di filtri o altre estensioni personalizzate.

4.2.1 Baseline con tomcat-users.xml

Creazione directory tree

Visto che tomcat permette di suddividere logicamente le web-app attraverso la definizione di 3 file di deploy diversi, possiamo facilmente raggiungere la baseline seguendo questa strada.

La definizione dei file deve seguire la struttura standard dei progetti web di Java, quindi si creano 3 progetti diversi:

```
admin/
    WEB-INF/
        web.xml
    admin.jsp
    login.html
    login-failed.html
devs/
    WEB-INF/
        web.xml
    devs.jsp
    login.html
    login-failed.html
sales/
    WEB-INF/
        web.xml
    sales.jsp
    login.html
    login-failed.html
```

Configurazione ruoli per ciascuna webapp

In ogni file web.xml è necessario inserire il ruolo che può accedere alla relativa webapp:

```
admin/WEB-INF/web.xml - <role-name>admin</role-name>
sales/WEB-INF/web.xml - <role-name>sales</role-name>
devs/WEB-INF/web.xml - <role-name>devs</role-name>
```

Nel file tomcat-users.xml è necessario specificare, per ciascun ruolo, gli utenti che lo hanno assegnato.

```
<tomcat-users>
    <role rolename="admin"/>
    <role rolename="sales"/>
    <role rolename="devs"/>
    <user username="filippo" password="filippo" roles="sales"/>
    <user username="sara" password="sara" roles="sales"/>
    <user username="lorenzo" password="lorenzo" \
                               roles="admin,devs,sales"/>
    <user username="andrea" password="andrea" roles="devs"/>
```

```
    <user username="antonio" password="antonio" roles="devs"/>
</tomcat-users>
```

4.2.2 Autorizzazione fine-grained utilizzando Policy CORS

La Cross-Origin Resource Sharing (CORS) è una politica di sicurezza dei browser web che regola come le risorse web vengano richieste da domini diversi rispetto a quello da cui la risorsa stessa è stata originata. La sua implementazione è strettamente collegata all'autorizzazione poichè determina quali origini esterne possono accedere alle risorse del server.
In Tomcat, le politiche CORS possono essere implementate utilizzando filtri, ovvero dei componenti Java che possono intercettare e manipolare le richeste e risposte HTTP. Di default Tomcat mette a disposizione una serie di filtri già definiti, tra cui il "CORS Filter". Il filtro funziona aggiungendo gli header Access-Control-* necessari all'oggetto HttpServletResponse secondo la specifica CORS del W3C. Se la richiesta è invalida o non è consentita, la richiesta viene respinta con il codice di stato HTTP 403 (Forbidden).
Supponiamo che la nostra web-app sotto l'endpoint admin/admin.jsp possa essere acceduta solo dalle richieste che hanno Header Origin https://admin.helloworld.com e che effettuano una richiesta POST. Allora il filtro da definire nel web.xml è:

```
<filter>
    <filter-name>CorsFilter</filter-name>
    <filter-class>
        org.apache.catalina.filters.CorsFilter
    </filter-class>
    <init-param>
        <param-name>cors.allowed.origins</param-name>
        <param-value>https://admin.helloworld.com</param-value>
    </init-param>
    <init-param>
        <param-name>cors.allowed.methods</param-name>
        <param-value>POST</param-value>
    </init-param>
    <init-param>
        <param-name>cors.allowed.credentials</param-name>
        <param-value>True</param-value>
    </init-param>
</filter>
<filter-mapping>
    <filter-name>CorsFilter</filter-name>
```

```
    <url-pattern>/admin/*</url-pattern>
</filter-mapping>
```

4.3 Autorizzazione in NodeJs

Node supporta sia ACL sia RBAC e sia ABAC, infatti, essendo un ambiente runtime versatile, permette di implementare tutti e 3 i modelli attraverso vari middleware e librerie.

4.3.1 Baseline con un file custom

Configurazione directory tree

NodeJs non si esprime riguardo al formato che l'applicazione javascript deve avere, significa quindi che possiamo creare una struttura personalizzata. Inseriamo all'interno della cartella public le risorse statiche, creiamo anche un file data.js che simula il file degli utenti.

```
home/
    app.js
    data.js
    public/
        index.html
        login.html
```

Contenuto del file data.js

NodeJs lascia molto spazio allo sviluppatore sul modo in cui l'autorizzazione può essere gestita. L'idea è quella di creare due array: il primo viene utilizzato per il lookup delle credenziali degli utenti, il secondo invece è l'array che specifica i gruppi, e per ogni gruppo gli utenti che gli appartengono:

```
const userDB = [
    {
        id: 1,
        user: "lorenzo",
        password: "$2a$10$njfsIgwi...quM8EVU.34y1l2",
    }
    ...
];
const groupDB = [
```

```
    {
        id: 1,
        name: "admin",
        users: [ "lorenzo" ],
    }
    ...
];
module.exports = { userDB, groupDB }
```

Contenuto del file app.js

All'interno del file dell'applicazione è necessario specificare il comportamento dell'endpoint raggiunto dopo il click sul bottone di submit nel form della pagina login.html. Si cerca di ottenere il gruppo al quale appartiene l'utente e in base a quello si costruiscono pagine diverse.

```
if (passwordMatch) {
    let foundGroup = findGroupByUsername(foundUser.user);
    if (foundGroup) {
        let response = `<h2>
            Ciao ${foundUser.user} appartieni a ${foundGroup}.
                        </h2>`;
        switch (foundGroup) {
            case 'sales':
                response = response.concat("Sales può vedermi");
                break;
            case 'devs':
                response = response.concat("Devs può vedermi");
                break;
            case 'admin':
                response = response.concat("Admin può vedermi");
                response = response.concat("Devs può vedermi");
                response = response.concat("Sales può vedermi");
        }
        res.send(response)
    }
}
```

La funzione findGroupByUsername() controlla all'interno dell'array groupDB specificato in data.js quale gruppo contiene lo username passato.

4.3.2 Controllo ABAC basato sulla data di accesso

A differenza dei modelli ACL e RBAC, che si basano principalmente su liste di controllo degli accessi e ruoli predefiniti, l'ABAC (Attribute-Based Access Control) permette di definire regole di accesso più dinamiche e flessibili utilizzando vari attributi dell'utente e del contesto. In questo caso, l'integrazione di una verifica della data di accesso consente di limitare l'accesso alle risorse a giorni specifici della settimana in base al gruppo di appartenenza dell'utente. Ad esempio, i membri del gruppo "sales" possono essere autorizzati ad accedere solo nei giorni lavorativi, mentre gli "admin" possono avere accesso tutti i giorni.

Funzione isAccessAllowed()

Si aggiunge una funzione per verificare se l'accesso è consentito in base alla data e al gruppo. Supponiamo che gli utenti del gruppo "admin" possono accedere alle risorse durante tutti i giorni della settimana mentre gli utenti dei gruppi "sales" e "devs" possono accedere alle risorse solo nei giorni feriali.

```
function isAccessAllowed(username, group) {
    const currentDate = new Date();
    const currentDay = currentDate.getDay();
    const accessRules = {
        admin: [0, 1, 2, 3, 4, 5, 6],
        sales: [1, 2, 3, 4, 5],
        devs: [1, 2, 3, 4, 5],
    };
    return accessRules[group] ?
        accessRules[group].includes(currentDay) :
        false;
}
```

La funzione sarà utilizzata con un'espressione di AND insieme al controllo foundGroup.

4.4 Autorizzazione in Nginx

Nginx è spesso utilizzato come reverse proxy e load balancer, dove le funzionalità avanzate di controllo degli accessi possono essere delegate ad altri componenti dell'infrastruttura, come applicazioni web o sistemi di gestione delle identità. Per questo motivo Nginx supporta nativamente solo il modello ACL attraverso l'utilizzo delle direttive Allow e Deny.

Precedentemente abbiamo visto che Nginx permette l'autenticazione tramite l'utilizzo di sottorichieste a server di autenticazione esterni, anche nel caso dell'autorizzazione è necessario un provider esterno che se ne occupa. Visto che il server di autenticazione Node aveva gestito molto bene la parte di autenticazione si andrà ad aggiornare la sua implementazione aggiungendo anche le funzionalità di autorizzazione, facendolo diventare parzialmente un server AAA (doppia A).

4.4.1 Baseline con la suddivisione dei token generati

Script authserver.js

Per gestire l'autenticazione si salvavano i token generati server-side in un array cookiesArray. Il token era salvato nel caso in cui le credenziali inviate dall'utente per autenticarsi combaciavano con quelle hardcodate nel main script js. Possiamo dire che formalmente è presente un solo ruolo (se così si può chiamare) e per questo non è stato applicato alcun "enforcement" relativo all'autorizzazione. Questo significa che l'utente autenticato è abilitato ad accedere a tutte le risorse sul webserver.
Per risolvere questo problema si creano tre array diversi (uno per ogni ruolo), e a seconda del gruppo al quale l'utente appartiene dopo essere stato autenticato si salva il relativo token generato nel relativo array. Questo permette di disaccoppiare il processo di autenticazione da quello di autorizzazione e di controllare la presenza dei token inviati dal client solo all'interno degli array del ruolo a cui appartengono veramente.

Struttura dati cookiesArray

Il cookiesArray si presenta come un dizionario contenente per ogni ruolo un array vuoto che verrà riempito a runtime.

```
let cookiesArray = [
    { name: "adminToken", array: [] },
    { name: "salesToken", array: [] },
    { name: "devsToken", array: [] },
]
```

Aggiunta della nuova funzionalità in /dologin

L'endpoint /dologin deve riconoscere a quale gruppo l'utente appartiene dopo averlo autenticato confrontandosi con il groupDB di data.js. Dopo aver identificato il gruppo inserisce il token generato nell'array relativo al suo gruppo:

```
if (passwordMatch) {
```

```
        let foundGroup = findGroupByUsername(foundUser.user);
        const randomString = generateRandomString(10);
        res.cookie('authCookie',
                    randomString,
                    {maxAge: 900000, httpOnly: true}
                  );
        switch (foundGroup) {
            case 'devs':
                cookiesArray[2].array.push(randomString);
                res.redirect('/protected/devs.html');
                break;
            case 'sales':
                cookiesArray[1].array.push(randomString);
                res.redirect('/protected/sales.html');
                break;
            case 'admin':
                cookiesArray[0].array.push(randomString);
                res.redirect('/protected/admin.html');
                break;
        }
    }
```

Aggiunta della nuova funzionalità in /auth

L'endpoint /auth deve confrontare il token inviato dal client con i token presenti negli array di token dei gruppi e stabilire a quale gruppo quel token appartiene. Inoltre deve controllare la risorsa richiesta dal client e controllare che il gruppo trovato possa accederci. Precedentemente abbiamo impostato Nginx in modo tale che quando invia la subrequest, per verificare se il client ha accesso alla risorsa, invia anche nell'header X-Original-URI il path della risorsa richiesta. Questo campo adesso ci torna molto utile per capire a quale risorsa l'utente sta tentando di accedere e quindi stabilire se esso ha i permessi necessari per l'accesso.

```
    app.get('/auth', (req, res) => {
        const cookieValue = req.cookies.authToken;
        const originalURI = req.get('X-Original-URI');
        let groupName = findGroupNameByToken(cookieValue);
        if (groupName) {
            switch(groupName) {
                case 'devsToken':
                    if (originalURI === '/protected/devs.html')
```

```
                    res.status(200).send();
                else
                    res.status(403).send();
                break;
            case 'salesToken':
                if (originalURI === '/protected/sales.html')
                    res.status(200).send();
                else
                    res.status(403).send();
                break;
            case 'adminToken':
                if (originalURI === '/protected/admin.html' ||
                    originalURI === '/protected/sales.html' ||
                    originalURI === '/protected/devs.html'
                    )
                    res.status(200).send();
                else
                    res.status(403).send();
                break;
        }
    }
})
```

La funzione findGroupNameByToken() controlla all'interno dell'array cookiesArray quale array contiene il token passato.

4.4.2 Filtrare il traffico basandosi sulla posizione geografica con GeoIP2

L'utilizzo della restrizione del traffico basato sulla posizione geografica è una misura di sicurezza avanzata che permette di limitare l'accesso a una rete o a un'applicazione web in base alla posizione geografica degli utenti. A livello Enterprise, si stabiliscono queste policy di autorizzazione per conformarsi a regolamenti locali.
Per implementare tale restrizione su Nginx, uno dei metodi più efficaci è l'utilizzo di GeoIP2, una libreria che consente di determinare la posizione geografica degli indirizzi IP in ingresso.
GeoIP2, sviluppato da MaxMind, fornisce database accurati e aggiornati che mappano gli indirizzi IP alle loro località geografiche corrispondenti. In Nginx, l'integrazione di GeoIP2 avviene tramite un modulo che consulta questi database per ogni richiesta in ingresso. Dopo aver determinato la posizione dell'utente, è possibile configurare Nginx per consentire o bloccare l'accesso basato su criteri geografici specifici. Ad esempio,

si può consentire l'accesso solo da determinati paesi o bloccare traffico proveniente da regioni note per attività dannose. Questa funzionalità si integra facilmente nel file di configurazione di Nginx, permettendo una gestione granulare e automatizzata delle politiche di accesso in base alla geolocalizzazione.
Supponiamo di voler bloccare l'accesso alle risorse quando la richiesta HTTP proviene da indirizzi IP mappati in Cina e in Russia.
Carichiamo in Nginx i moduli necessari per importare GeoIP2:

```
load_module modules/ngx_http_geoip2_module.so;
load_module modules/ngx_stream_geoip2_module.so;
```

Specifichiamo, nel blocco http, il percorso al file database di GeoIP2 scaricato seguendo la guida sul sito di Nginx, impostiamo delle variabili custom valorizzate con i dati ritornati dal database e infine aggiungiamo la direttiva map che permette di filtrare il traffico secondo la policy appena stabilita.

```
http {
 geoip2 /usr/share/GeoIP/GeoLite2-Country.mmdb {
      $iso_code country iso_code;
 map $iso_code $allowed_country {
       default yes;
       CN no;
       RU no;
}
```

4.5 Autorizzazione in Weblogic

Weblogic supporta le liste di controllo degli accessi (ACL), consentendo la configurazione di permessi dettagliati per utenti e gruppi specifici. Inoltre, Weblogic ha un robusto supporto per il controllo degli accessi basato sui ruoli (RBAC). RBAC in Weblogic permette di assegnare permessi a ruoli specifici e di collegare gli utenti a questi ruoli. Infine, Weblogic supporta il controllo degli accessi basato sugli attributi (ABAC), permettendo la definizione di politiche di accesso basate su attributi complessi degli utenti e delle risorse. La capacità di Weblogic di integrare ACL, RBAC e ABAC lo rende una soluzione potente e versatile per la gestione degli accessi in ambienti complessi, dove è necessario bilanciare sicurezza, flessibilità e facilità di gestione.
Come per Tomcat anche in WebLogic è necessario definire tre webapp indipendenti per suddividere le applicazioni web accedibili da ciascun ruolo definito. Ogni applicazione ha bisogno di definire all'interno del proprio file web.xml quali ruoli possono accedere. Inoltre weblogic mette a disposizione un ulteriore file: weblogic.xml, che permette di mappare ad ogni ruolo una serie di principal name.

4.5.1 Baseline con il file weblogic.xml

Ogni applicazione deve contenere un file weblogic.xml all'interno della cartella standard WEB-INF, dove definiamo i principal-name bindati ad uno specifico ruolo per quella determinata web-app, quindi:

```
admin/WEB-INF/weblogic.xml :
    <role-name>user</role-name>
    <principal-name>admin</principal-name>
sales/WEB-INF/weblogic.xml :
    <role-name>user</role-name>
    <principal-name>sales</principal-name>
    <principal-name>admin</principal-name>
devs/WEB-INF/weblogic.xml :
    <role-name>user</role-name>
    <principal-name>devs</principal-name>
    <principal-name>admin</principal-name>
```

Il ruolo user è comune a tutte le 3 web application ma i principal user bindati ad esso sono diversi e sono assegnati secondo la policy descritta dalla Baseline. In questo caso i principal-name sono i gruppi definibili dalla console di weblogic.
Di conseguenza in ogni file web.xml è necessario definire il role-name user:

```
<role-name>user</role-name>
```

Conclusione

Nel corso della tesi, è stato dimostrato come l'adozione di protocolli avanzati e tecniche di sicurezza possa migliorare significativamente la protezione dei dati nelle comunicazioni web. Attraverso l'analisi di strumenti come OpenSSL e l'implementazione di soluzioni basate su certificati digitali e infrastrutture a chiave pubblica (PKI), è stata evidenziata l'importanza di un approccio olistico alla sicurezza. Le metodologie di autenticazione e autorizzazione approfondite hanno mostrato come sia possibile ridurre i rischi di accessi non autorizzati e migliorare l'integrità e la confidenzialità dei dati.
Ci sono però diverse aree che meritano ulteriori indagini e miglioramenti e possono essere spunti utili per sviluppare ricerche future. Ecco alcuni consigli:

- Automazione della sicurezza: Implementazione di sistemi di sicurezza automatizzati che possano adattarsi dinamicamente alle nuove minacce.
- Tecniche di autenticazione avanzate: Esplorazione di nuove metodologie di autenticazione che utilizzano biometria e altre forme di verifica multi-fattore.
- Miglioramento delle infrastrutture a chiave pubblica: Sviluppo di soluzioni PKI più robuste e scalabili per una gestione dei certificati ancora più efficiente.
- Sicurezza basata su intelligenza artificiale: Utilizzo di algoritmi di intelligenza artificiale per rilevare e rispondere a minacce in tempo reale.
- Protocolli di sicurezza quantistica: Ricerca e sviluppo di protocolli che possano resistere alle capacità di calcolo dei futuri computer quantistici.

La sicurezza dei sistemi web rimane una sfida in continua evoluzione. Le soluzioni proposte e analizzate in questa tesi rappresentano un passo avanti significativo, ma la ricerca deve continuare per fronteggiare le nuove minacce e garantire un livello di sicurezza sempre più elevato.

Bibliografia

[1] Alan O. Freier, Philip Karlton, and Paul C. Kocher. The Secure Sockets Layer (SSL) Protocol Version 3.0. RFC 6101, August 2011.

[2] Eric Rescorla and Tim Dierks. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246, August 2008.

[3] Eric Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446, August 2018.

[4] Scott Helme. Top 1 million analysis, June 2022. `https://scotthelme.co.uk/top-1-million-analysis-june-2022/`.

[5] Agenzia per l'Italia Digitale. Raccomandazioni agid in merito allo standard transport layer security (tls). `https://cert-agid.gov.it/wp-content/uploads/2020/11/AgID-RACCSECTLS-01.pdf`.

[6] CVE-2012-4929. Available from MITRE, CVE-ID CVE-2012-4929., 2012.

[7] CVE-2014-0160. Available from MITRE, CVE-ID CVE-2014-0160., 2014.

[8] OASIS Security Services TC. Security assertion markup language (saml) v2.0 technical overview, March 2008. `https://docs.oasis-open.org/security/saml/Post2.0/sstc-saml-tech-overview-2.0.html`.

[9] Michael B. Jones, John Bradley, and Nat Sakimura. JSON Web Token (JWT). RFC 7519, May 2015.

[10] Michael B. Jones, John Bradley, and Nat Sakimura. Final: Openid connect core 1.0 incorporating errata set 2, December 2023. `https://openid.net/specs/openid-connect-core-1_0-errata2.html`.

[11] Michael B. Jones and Dick Hardt. The OAuth 2.0 Authorization Framework: Bearer Token Usage. RFC 6750, October 2012.

[12] FIDO Alliance. Fido 2.0: Client to authenticator protocol, October 2017. https://fidoalliance.org/specs/fido-v2.0-rd-20170927/fido-client-to-authenticator-protocol-v2.0-rd-20170927.html.

[13] Meta DB. The web servers (http daemons) most used to serve web pages, 2024. https://metadb.co/en/most-used-web-server-daemons.

[14] Kathleen Moriarty, Magnus Nyström, Sean Parkinson, Andreas Rusch, and Michael Scott. PKCS #12: Personal Information Exchange Syntax v1.1. RFC 7292, July 2014.

[15] Shubham Munde. Identity and access management (iam) market overview, March 2024. https://www.marketresearchfuture.com/reports/identity-access-management-market-2635.

[16] Burt Kaliski. Public-Key Cryptography Standards (PKCS) #8: Private-Key Information Syntax Specification Version 1.2. RFC 5208, May 2008.

[17] Magnus Nyström and Burt Kaliski. PKCS #10: Certification Request Syntax Specification Version 1.7. RFC 2986, November 2000.

[18] Sharon Boeyen, Stefan Santesson, Tim Polk, Russ Housley, Stephen Farrell, and David Cooper. Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. RFC 5280, May 2008.

[19] Stefan Santesson, Michael Myers, Rich Ankney, Ambarish Malpani, Slava Galperin, and Dr. Carlisle Adams. X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP. RFC 6960, June 2013.