ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA

SCUOLA DI INGEGNERIA E ARCHITETTURA
DIPARTIMENTO DI INFORMATICA – SCIENZA E INGEGNERIA - DISI
CORSO DI LAUREA MAGISTRALE IN INGEGNERIA INFORMATICA

TESI DI LAUREA
in
Laboratorio Di Sicurezza Informatica T

Analisi dei vettori di attacco alle architetture a Microservizi e definizione di strategie di contrasto per i Blue Team

CANDIDATO
FEDERICO GALANTI

RELATORE
Chiar.mo Prof.
MARCO PRANDINI

CORRELATORI
Prof. Andrea Melis
Dott. Davide Berardi

Anno Accademico 2019/2020

Sessione III

Ringraziamenti

Al prof. Prandini, al prof. Melis e al dott. Berardi, che per la seconda volta mi hanno dato la possibilità di concludere la mia carriera universitaria, sostenendo la validità delle mie tesi di laurea e credendo nel lavoro qui presentato.

Ai professori tutti del Corso di Ingegneria Informatica, con particolare menzione alla prof.ssa Mello, al prof. Chesani, al prof. Denti e al prof. Bellavista, per avermi non solo formato e temprato nelle competenze e nell'arte informatica, ma anche come professionalità e persona.

Alla mia famiglia, soprattutto mio padre, Sabatino, e mia madre, Laura, che mi ha sostenuto ed accompagnato in questo lungo percorso, donandomi la forza di concludere finalmente il capitolo universitario della mia vita.
A loro nuovamente, per aver creduto nelle mie potenzialità e per avermi permesso di iniziare e completare il mio percorso universitario.

Ai miei amici, Matteo, Alessandra, Simone, Niccolò e Carlotta, che mi hanno aiutato, spronato, sostenuto e hanno contribuito a rendermi la persona che è riuscita a chiudere questo percorso con orgoglio ed onore, con la loro presenza e il tempo concessomi.

Infiniti ringraziamenti. Il coronamento di questo sogno, la fine di questo percorso. Grazie.

Abstract

Con la diffusione delle nuove architetture di sviluppo software, come Service-Oriented Architecture e Microservice, l'applicazione web cessa di essere un semplice programma su un server, ma diventa sempre più simile ad una rete di applicazioni che cooperano tra loro. Attualmente le minacce informatiche si fanno sempre più aggressive e feroci. Per evitare di diventare vittime in questo cambiamento architetturale, è necessario fornire conoscenza, metodi e strumenti per rendere sempre più efficaci i team di sicurezza informatica e scongiurare possibili minacce. Questa tesi rappresenta un documento di partenza per fornire metodi e strumenti adeguati per un Blue Team, i difensori. A questo proposito, verrà prima presentato lo stato dell'arte dell'architettura Microservice, con le tecnologie e la conoscenza alla base. Successivamente, si analizzeranno le vulnerabilità, le criticità e i vettori d'attacco a architetture a microservizi, e le differenze, in termini di sicurezza, con un'architettura monolitica, al fine di individuare una strategia generale con cui affrontare l'argomento di difesa di un'architettura a microservizi.

Sommario

1. Introduzione

1.1. Motivazioni della tesi

Se nella vita di tutti i giorni, la digitalizzazione sta facendosi strada in maniera sempre più preponderante, ci sono cambiamenti al di sotto della pelle della tecnologia stessa che stanno rimodellando come la tecnologia viene costruita. Siamo passati dai personal computer, ai laptop, ai tablet, poi agli smartphone e alle tecnologie wearable, come gli smartwatch, che stanno diventando sempre più diffuse.

Questa è tuttavia la superficie. Il mondo digitale, in questo momento, vive di estremo fermento e di rivoluzioni. Il cancello d'ingresso è Industria 4.0, già standardizzata nel RAMI. Ma non c'è solo una rivoluzione nel modello dell'industria e dell'impresa in generale: anche il modo in cui sono progettati i moderni sistemi informatici sta cambiando. Il paradigma cloud computing ha avviato questa rivoluzione, in cui le Web Application (o in generale, i servizi informatici distribuiti) adesso non sono più replicate su tanti server fisici, ma sono sulla piattaforma cloud, che le rende facilmente distribuibili e accessibili ovunque. Il cloud computing è il "pioniere" di queste tecnologie, sono poi succeduti l'Edge e il Fog computing, come complementi per una rete sempre più distribuita e ramificata, senza avere sistemi centralizzati.

Questa è la chiave della rivoluzione informatica attuale, la volontà di de-centralizzare quanto più possibile, per avere un sistema "veramente distribuito".

Il cloud continuum (così è chiamato l'insieme Cloud-Fog-Edge) non basta però. Serve qualcosa di più incisivo, perché sebbene i mezzi di distribuzione siano diventati distribuiti e sempre più decentralizzati, qualcosa di più è necessario. L'attuale richiesta di servizi e l'asta della qualità di servizio impostata dalle grandi aziende del settore IT (Microsoft-Apple-Google-Amazon), vuole che i web services siano più rapidi, sempre a disposizione, con elevati standard di affidabilità e disponibilità. Questi requisiti non influenzano solo il servizio in sé, ma tutta la catena del servizio: dalla progettazione e sviluppo, fino al deployment e alla manutenzione, si vuole garantire quanto più possibile disponibilità, limitando i downtime e permettendo di effettuare operazioni sui web services senza necessità di mandare offline i propri servizi per potenziarli o modificarli, in risposta a problemi o per aggiungere funzionalità. La rivoluzione sta nel mettere in dubbio l'architettura monolitica stessa delle applicazioni.

La modularizzazione ha provato a sfidare i limiti dell'architettura monolitica, semplificando e settorializzando il codice, in modo da garantire una più facile manutenibilità, leggibilità e sviluppo del codice, rendendo l'architettura monolitica più flessibile e facilmente modificabile, ma sempre che siamo in una "unica applicazione". Un macro blocco, in cui se qualcosa non funziona, che sia modulare o meno, tutto non funziona. Un macro blocco che, se va modificato, ma comunque modificato tutto, in quanto non è possibile semplicemente "separare il modulo" e poi re-inserirlo nel "muro" una volta che è stato modificato o separato. Questo perché quei moduli sono strettamente accoppiati l'uno con l'altro, come se fossero costruiti uno sopra all'altro e cementificati da codice statico.

Microservice Architecture. Due parole, per indicare una rivoluzione senza precedenti. Non più "un'applicazione", ma "tanti servizi" che, collegati tra loro opportunamente, formano "applicazioni". Un approccio in cui le funzionalità sono raggruppate per business logic, vengono costruite in "servizi", delle "micro-applicazioni" separate, distinte e disaccoppiate, che possono collaborare scambiandosi messaggi per ottenere il comportamento di un'applicazione o di una Web app o di un Web Service. Adesso i servizi non dipendono tra loro, possono essere tranquillamente manutenuti senza che tutto vada giù ed errori o problemi non fanno andare giù tutto il "muro" dell'applicazione. E si può sviluppare continuamente, ottenendo performance, precisione, affidabilità e disponibilità senza precedenti.

Come una rivoluzione si mette in moto da una parte, la rivoluzione avviene anche nel crimine. WannaCry, Stuxnet, Zeus, Emotet, sono solo alcuni nomi che hanno inspirato terrore e danni incalcolabili negli ultimi anni. La minaccia non è più quella di "entrare nella rete del Pentagono per leggere quei file segreti che nessuno vuole che si conoscano". Non c'è più il complotto. C'è il guadagno e questo profitto è dato da segreti industriali, ma anche da dati, danni creati ad un'impresa che corrispondono a guadagni di un'altra. Però i bersagli sono diventati anche ospedali, enti governativi, enti finanziari e privati. Il pericolo è diventato molto più grande.

E' imperativo che una rivoluzione informatica e digitale debba essere accompagnata da un'evoluzione nella cybersecurity, senza compromessi. E' imperativo che la cultura della sicurezza informatica si diffonda, che la sicurezza non si fermi più solo ad aspetti di difesa dall'esterno, ma anche dall'interno, fino a scuotere nelle fondamenta la definizione stessa di "fiducia" in una rete.

Chi pensa sia un lontano futuro si sbaglia. I Microservizi stanno proliferando da un po': già nel 2015, le aziende IT si preparavano alla rivoluzione dei Microservizi, come dimostrato da un sondaggio NGINX (1) in cui il 68% delle imprese era già al lavoro sullo studio o sull'implementazione dell'architettura a microservizi. Si stima che per il 2023, questo approccio di sviluppo diventerà uno standard, soppiantando l'architettura monolitica, arrivando ad un valore di mercato pari a 32 miliardi di dollari (US) (2).

1.2. Obiettivo

L'obiettivo di questa tesi è quindi investigare l'architettura dei Microservizi e scandagliarne le insidie e gli aspetti di sicurezza, dal punto di vista del "Blue Team".

L'evoluzione del mondo digitale, nel tempo, ha corrisposto un'evoluzione nel mondo informatico e dello sviluppo. Se prima, nel panorama più generale, il processo della sicurezza informatica era principalmente reattivo e si svolgeva solo a fronte di problematiche o segnalazioni delle autorità (IEFT, NIST, agenzie governative, etc…), adesso assume una forma più sistematica e formale. In particolare, non si pensa più ad un processo strutturato nelle fasi di scoperta e di chiusura della vulnerabilità, esclusivamente mirato a "chiudere la vulnerabilità", ma si sviluppa in un processo a più fasi, con al centro la simulazione di attacchi informatici al fine di individuare non solo la vulnerabilità, ma le dinamiche che possono capitare e l'insieme degli eventi che possono indicare problematiche. Una simulazione d'attacco comporta il testare le difese e la coesione del sistema nei confronti di minacce esterne e, in quanto simulazione, necessita non solo di un attaccante, ma anche di un difensore, per rendere ancora più efficace la raccolta di informazioni su ogni aspetto della sicurezza informatica. Le parti sono individuate come la Squadra Rossa (Red Team) e la Squadra Blu (Blue Team), che si "sfidano" per trovare vulnerabilità (la prima) e per imparare a difendersi dagli attacchi informatici (la seconda). E' così che sono definiti i team che si occupano di Vulnerability Assessment e Penetration Testing (Red Team) e Cybersecurity (Blue Team).

L'ambito specifico in oggetto è quello di provare ad indicare un accenno di modello con cui sia possibile assistere il Blue Team nelle operazioni di difesa, in un'architettura a Microservizi. Premettendo che non esiste ancora un approccio olistico alla sicurezza dei Microservizi, l'obiettivo finale è porre a confronto le architetture monolitica e microservizi dal punto di vista della difesa dagli attacchi informatici, cercando di rispondere a quanto segue:

1) Quanto è veramente estesa la superficie d'attacco in un sistema a Microservizi?

2) E' possibile, a partire da un modello di architettura a Microservizi e ricordando quanto già esiste per l'architettura monolitica, individuare una strategia difensiva applicabile a tutta l'architettura (Blue team strategy)?

3) Sono presenti strumenti efficaci per le operazioni di individuazione, contenimento, espulsione e stima dei danni in caso di un avversario già all'interno di un componente della nostra architettura?

4) E' possibile ricavare quantomeno considerazioni generali, da applicare a possibili modelli difensivi per architetture a Microservizi? Esistono possibili soluzioni?

2. Microservizi – Background e stato dell'arte

2.1. Introduzione ai Microservizi

I Microservizi (o Architettura MicroService) sono un modello architetturale di sviluppo di piattaforme di servizio e sono l'evoluzione dell'Architettura Service-oriented (3). Le Web applications sono formate da una collezione di "servizi" a granularità-fine, ossia delle "mini applicazioni" specializzate in determinati tipi di operazioni o ambiti operativi, che comunicano tra loro attraverso protocolli di comunicazione leggeri, cooperando per ottenere il risultato previsto dall'applicazione specifica. La granularità fine è richiesta dall'approccio Domain-Driven Design, per cui si vuole sviluppare componenti specifici per il dominio applicativo ed eventualmente quanto più **riusabili possibile** (permettendo un guadagno non indifferente in termini sia di efficienza che di agilità nello sviluppo). I servizi in questione sono visti come componenti lascamente accoppiati, che concorrono singolarmente alle funzionalità anche di più applicazioni allo stesso tempo, fornendo le loro operazioni (3). Per essere più precisi, i Microservizi:

- Sono rilasciabili (deployment) in maniera indipendente e come componenti "autonome".
- Sono tecnologicamente indipendenti: non è necessario che usino protocolli di comunicazione specifici, sono indipendenti da **linguaggio di programmazione**, **database**, **hardware** e **ambiente software**.
- Sono di piccole dimensioni, compatibili con la comunicazione a scambio di messaggi, legati al proprio contesto d'esecuzione e **decentralizzati**.
- Lo sviluppo e il deployment sono autonomi, ossia non dipendono da altri servizi o componenti, e automatizzati.

Come per SOA, questo approccio nasce da problemi attuali dello sviluppo software e dell'avanzamento tecnologico: le tecnologie Cloud permetto di astrarsi dai vincoli delle macchine fisiche nello sviluppo di software, permettendo la "distribuzione" dei componenti applicativi.

L'architettura monolitica presenta forti problemi di scalabilità all'aumentare della complessità dell'applicativo. Gli attuali servizi (e-commerce, motori di ricerca, streaming di dati multimediali, et cetera) sono diventati estremamente complessi e non hanno più una struttura interna "intuitiva" o leggera, anche se sono costruite modularmente.

I tipi di tecnologie sono diventati così tanti che lo sviluppo "compatibile" richiede sforzi così grandi che sono lesivi, se comparati con gli attuali benefici. Si pensi al fatto che attualmente si sviluppano app per dispositivi mobili in due linguaggi di base (Swift e Java) per due sistemi operativi di base (iOS e Android) su multiple architetture hardware diverse.
Si vorrebbe poter "astrarre" dalla tecnologia in maniera più marcata, rispetto all'uso dei pattern, costruendo una volta sola il componente software necessario, per poterlo distribuire e farlo scalare con qualsiasi tecnologia.

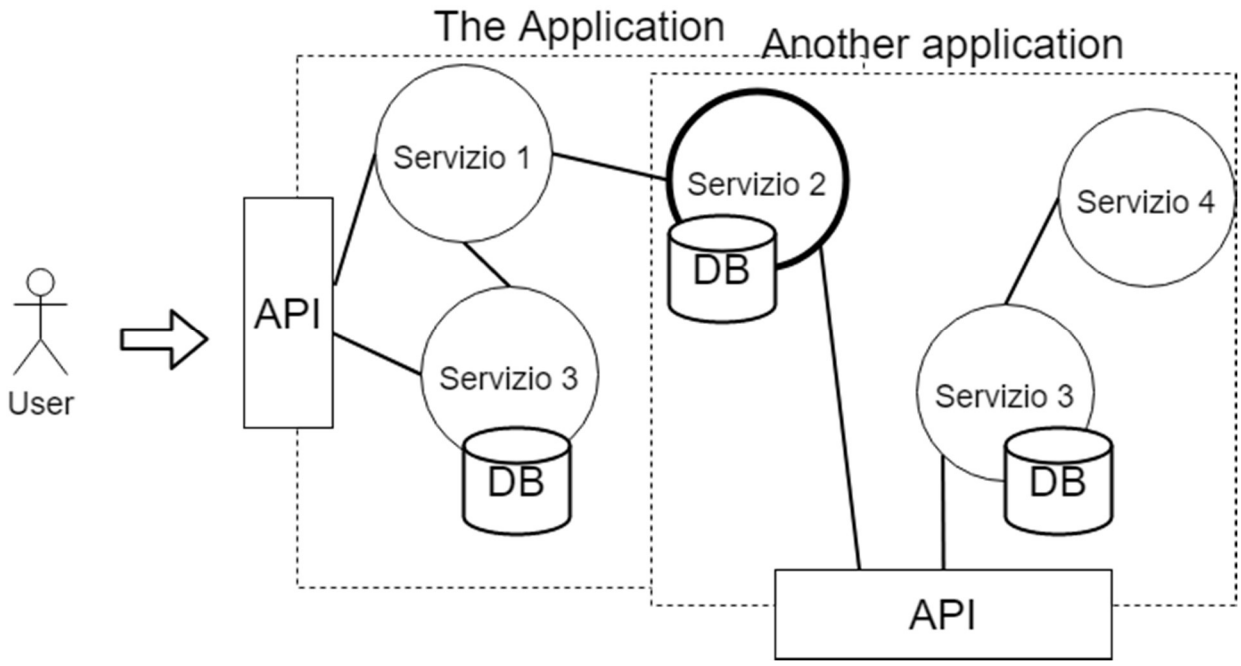

Figura 1 – Architettura microservizi: i servizi sono delle unità di logica di business che possono essere interconnesse tra loro per ottenere un'applicazione. In immagine, il "servizio 2", ad esempio, può far parte di "più applicazioni". Questo rende i servizi estremamente flessibili e potenti. Il componente "API" o API Gateway è il componente con cui l'utente può interagire con l'applicazione: l'utente invia richieste, che vengono inoltrate ai servizi, i servizi forniscono risposte che vengono poi date all'utente.

2.2. Retrospettiva: l'architettura monolitica

Per comprendere meglio, partiamo dal "come" sono costruite le applicazioni, comprese le Web Applications. Ossia, l'architettura monolitica. Solitamente, un'applicazione, qualsiasi sia la caratterizzazione o il campo, può essere immaginata come un "insieme di funzionalità": quando scriviamo il programma, abbiamo le funzioni, abbiamo i componenti e dobbiamo cablare insieme funzioni e componenti in maniera da ottenere quello che l'applicazione deve fare per noi. Tutto è contenuto in questa "applicazione": quelle che apparentemente sono "funzionalità", non sono "parti esterne" dell'applicazione, sono tutte contenute all'interno dell'applicazione. Come risultato, si ha un'applicazione compatta che, per essere modificata, deve essere completamente fermata, modificata e riavviata. Ancor peggio: se un componente non funziona bene, una funzione, una variabile è scritta male, l'intera applicazione può non funzionare a prescindere.

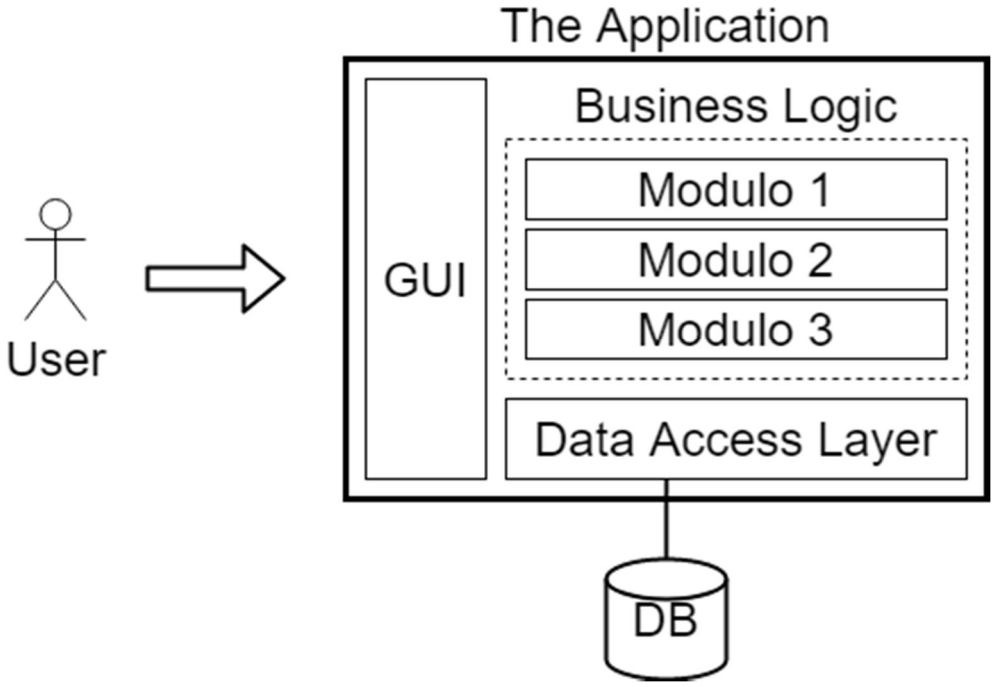

Figura 2 – Applicazione nell'architettura monolitica. Mentre prima l'utente interagiva solo con l'API Gateway, qui la GUI è interna all'applicazione, quindi l'utente interagisce con l'intera applicazione stessa.

Questo è estremamente poco scalabile, soprattutto nell'ambito attuale dello sviluppo software distribuito: come abbiamo detto, l'applicazione è una e in un "blocco unico". Immaginiamo dunque di voler avere una web application, su più server, per permettere la fruizione ad un pubblico quanto più vasto possibile. Se abbiamo "una sola unica applicazione", l'unica cosa possibile sarà produrre repliche: ossia per ogni macchina fisica, si avrà una copia dell'applicazione.

Inoltre, dovremo mantenerla: l'applicazione verrà modificata, andrà aggiornata, le macchine su cui si trova l'applicazione andranno protette e qualsiasi aggiornamento deve essere propagato su tutte le copie. Oltre a questa propagazione, bisogna contare quanto detto prima, ossia che l'applicazione va fermata, aggiornata e riavviata. Questo comporta un downtime, ossia un periodo di tempo per cui il servizio non è disponibile presso quella macchina che sta avendo l'applicazione aggiornata. Uno dei problemi dei downtime è che, più questi sono lunghi, più questi comportano un danno economico, perché sono "servizi non erogati". Per capire meglio l'entità di questo danno, gli "hacker", o terze parti malevole nel mondo dell'informatica, hanno incrementato gli attacchi di DoS (Denial of Service), proprio perché, in assenza della possibilità di estorcere denaro tramite Ransomware, in questo modo si assesta a priori un danno economico per il proprietario del sistema vittima.

Si consideri il caso del 28 Febbraio del 2017, in cui l'AWS S3 Service (Simple Storage Service) di Amazon è andato offline per la regione a nord della Virginia (4): secondo stime effettuate da Cyence, una compagnia di stima dei danni dovuti ad attacchi informatici, il downtime di questi servizi ha causato 150 mln di dollari di perdite per più di 500 aziende e 160 mln per i servizi finanziari del Stati Uniti (5). L'incidente ha avuto durata di 4 ore ed è stato causato, secondo quanto comunicato, da un errore umano dovuto, in fase di debug, all'inserimento di un input incorretto durante l'inserimento di un comando per rimuovere una piccola porzione di servers dal sottosistema S3. Quindi, anche i downtime da aggiornamento dei servizi sono un problema, in quanto il danno non era relativo all'errore in sé, ma al fatto che l'intero sistema S3 per quella regione fosse "non disponibile".

Un'evoluzione iniziale dell'architettura monolitica, atta proprio a sveltire i processi di sviluppo e deployment delle applicazioni, è quella della modularizzazione: in fase di progettazione, si sviluppa il cuore dell'applicazione e si predispongono meccanismi per far aggiungere funzionalità come moduli. Il vantaggio è che un modulo può essere tolto o aggiunto e questo non richiede la modifica di "tutto" il codice dell'applicazione. Ma è un palliativo, in quanto si tratta sempre di "un unico blocco d'applicazione" e soffrirà degli stessi problemi della normale architettura monolitica per quanto riguarda l'ambiente distribuito.

C'è un'ulteriore problema che però va considerato, legato alla tecnologia: ossia la dipendenza tecnologica stessa delle applicazioni. Le applicazioni, qualsiasi esse siano, sono scritte in determinati linguaggi e, di conseguenza, poggiano su definite architetture. Questo comporta che, ad esempio, una stessa applicazione debba essere scritta in più linguaggi diversi, dato che, sul mercato, sono presenti dispositivi diversi, con ambienti di esecuzione, sistemi operativi ed anche architetture hardware diverse. L'utilizzo di framework e middleware, a livello di sviluppo, può aiutare non poco a venire incontro alle esigenze di interoperabilità e di disaccoppiamento con le tecnologie specifiche, ma poi l'applicazione diventa dipendente anche dal middleware e/o dal framework stesso, riportandoci al punto di partenza.

Quindi, in sintesi, l'architettura monolitica presenta forti limiti:

- **Scalabilità**: ogni volta che qualcosa di nuovo va aggiunto o qualcosa va modificato, bisogna fare una "nuova" versione dell'applicazione, testarla, spegnerla se è a disposizione come Web Application o servizio distribuito, aggiornarla e riavviarla. In più, ogni modifica non si propaga automaticamente, ma bisogna propagare l'applicativo stesso (o le parti interessate, con il patching) alle copie per procedere all'aggiornamento.
- **Agilità**: un problema derivante dal problema di scalabilità. Non è modificabile nemmeno per modifiche contenute ed immediate. Bisogna rifare tutta l'operazione di building e deploying.
- **Dipendenza tecnologica**: spesso si vorrebbero utilizzare più linguaggi, sfruttando le potenzialità di ogni linguaggio o cercando di raggiungere tutta la platea di dispositivi sul mercato. Invece, ogni "applicazione" è legata ad un linguaggio, od un insieme di essi, fisso e limitato.
- **Tolleranza ai guasti**: se anche solo una feature, una funzione, o un modulo non va bene, tutta l'applicazione non va bene e il servizio non funzionerà, bisognerà terminare l'applicazione, trovare il problema, testarla e, in caso di esito positivo, riavviarla.

2.3. Attualmente: dall'architettura monolitica alla Service Oriented Architecture

Mentre da un lato l'architettura monolitica impedisce di attuare modifiche ad un software e di propagarle, da un lato, nel mondo dello sviluppo software, si cerca di rendere più efficiente il lavoro dietro il ciclo di vita di un'applicazione, specie in termini di servizi distribuiti. Il ciclo di vita di un'applicazione si distingue in due macrofasi:

- **Sviluppo**: che va dalla fase di progettazione alla fase di implementazione.
- **Manutenzione**: il lavoro post-rilascio, per assicurarsi il funzionamento continuo del prodotto.

Il problema è che la manutenzione, spesso e volentieri, finisce per diventare un'operazione di "sviluppo" continuo: durante l'operatività, si individuano problemi, si trova una soluzione, si applica. Oppure, si scopre o si individua la necessità di nuove funzioni, che andranno implementate. Mentre precedentemente non c'era molta "interattività" tra il cliente e il produttore, adesso il cliente si trova ad esprimere più facilmente e rapidamente le proprie necessità o le problematiche del prodotto. Inoltre, fattori esterni possono richiedere modifiche tempestive al prodotto stesso: si pensi al caso della crescita nella preoccupazione della privacy, gli incidenti relativi alla privacy, all'uso scorretto dei dati personali e le relative legislazioni al riguardo, come ad esempio il GDPR.

Questi eventi hanno introdotto l'obbligo di utilizzare determinate pratiche ed accorgimenti nella struttura e nell'utilizzo delle applicazioni e tale adeguamento doveva avvenire quanto più velocemente possibile. La velocità, ora, è fondamentale, per quanto riguarda il settore dei servizi IT.

Abbiamo quindi processi di sviluppo continui e velocità, come requisiti per un servizio informatico. Oltre a ciò, l'attuale struttura di un servizio informatico deve reagire od agire velocemente. Normalmente, le componenti dietro ad un'applicazione e al suo rilascio sul mercato come servizio, a livello di lavoro, sono:

- **Sviluppo**: il team che si occupa di sviluppare il software e le sue componenti.
- **Operations**: i servizi e i processi informatici offerti dall'azienda verso clienti interni o esterni. In larga parte, il supporto IT agli utenti.
- **QA**: o Controllo Qualità, che si adopera al testing e alla verifica dei requisiti richiesti o desiderati dagli stakeholder e dai clienti del progetto software.

Queste tre componenti del reparto IT sono normalmente separate e questo rallenta di molto il processo di sviluppo, sia iniziale che successivo, del software. Per risolvere quindi il problema di velocità, nascono gli approcci DevOps (6). Ma non basta, perché il problema è insito nell'architettura monolitica e nella sua scarsa flessibilità.

Quindi si passa alla Service-Oriented Architecture (7): è uno stile architetturale di design software, in cui le varie funzionalità (o i moduli di un'applicazione, come parallelismo all'architettura monolitica) vengono raccolte in componenti applicativi, chiamati servizi (8). Quando un servizio deve essere fornito, il sistema interroga il componente applicativo necessario e ottiene la risposta da ritornare all'utente. L'interrogazione del componente di servizio al componente applicativo avviene tramite comunicazione attraverso una rete. Queste unità di business logic sono i "servizi" e hanno le seguenti proprietà:

- Implementano business logic secondo lo specifico risultato che devono ritornare, se interrogati.
- Sono auto-contenuti e indipendenti tra loro.
- Dal punto di vista dell'utente sono viste come black box. Loro fanno una richiesta e vedono il risultato, ma non vedono il loro funzionamento
- Possono essere composti a loro volta da altri servizi.

L'applicazione in sé utilizza questi servizi per servire le funzionalità richieste dal cliente, ossia l'interfaccia verso l'esterno, genericamente esposta in forma di API. A loro volta, i servizi possono comunicare tra loro attraverso interfacce di comunicazioni simili. Una forma del modello Service-Oriented Architecture altro non è che i Microservizi (9).

2.4. Microservizi – Struttura e funzionamento

Per rendere ancora più chiaro cosa siano i Microservizi: immaginiamo dunque l'architettura monolitica come una costruzione di mattoncini, che ne rappresentano i moduli. Ogni mattoncino contiene funzionalità che sostengono la costruzione. I mattoncini sono legati tra loro con il cemento, quindi sono un tutt'uno. Se un mattoncino si rompe o manca, manca una parte che sostiene il muro e il muro crolla sotto il suo peso.

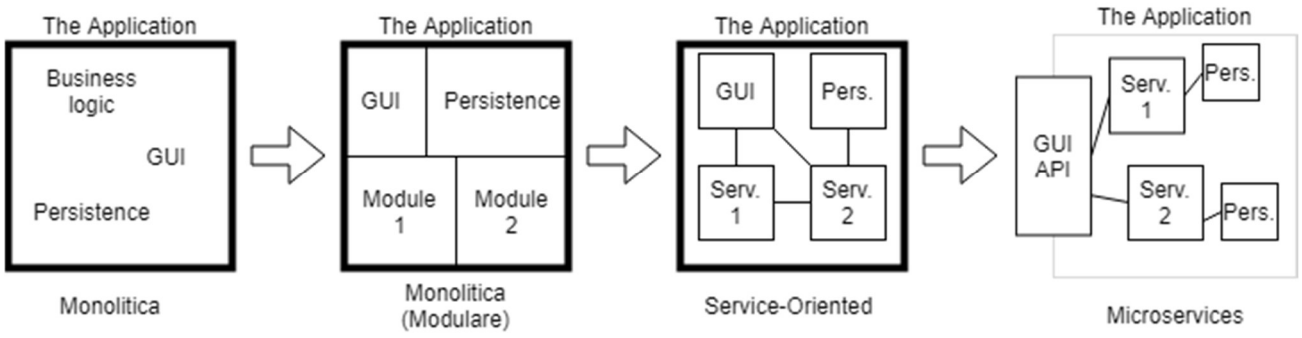

Figura 3 – L'evoluzione dall'architettura monolitica, in cui si ha un unico grande programma, alla struttura modulare, in cui si "spezzetta" il codice in più moduli, alla formazione di moduli intesi come servizi.

L'architettura a Microservizi, invece, come la SOA, si può pensare come le costruzioni a blocchi magnetici. I blocchi delle strutture sono le sfere e le barre che li collegano sono canali di comunicazione basati su scambio di messaggi. Su diverse macchine o in un contesto unico (rappresentato dall'applicazione), quindi, più servizi specifici possono convivere contemporaneamente, slegati l'uno all'altro, ma allo stesso tempo uno al servizio degli altri. E' immediato capire che, rispetto all'architettura monolitica:

- Ogni servizio (blocco) è a sé stante. Può essere tolto, maneggiato e rimesso senza che l'applicazione smetta di funzionare. Al più, solo quelle funzionalità saranno temporaneamente inaccessibili.
- Ogni servizio può essere trasferito in maniera agevole, grazie al paradigma a "container", e tramite l'orchestrazione, se un servizio dovesse non essere disponibile, l'orchestratore ne attiva una copia immediatamente. Questo aiuta enormemente anche in fase di manutenzione e aggiornamento, rendendo praticamente inesistenti i downtime.
- Poiché ogni servizio è a sé stante e per usufruirne basta sapere come accedere alla sua interfaccia, ogni servizio può essere scritto in qualsiasi linguaggio e usare qualsiasi tipo di tecnologia, poiché l'importante è che sappia come comunicare (HTTP può essere adattato facilmente).

Ogni servizio è scalabile a piacimento, senza che questo vada a impattare sull'intera struttura delle applicazioni. Un servizio può essere modificato e ampliato, anche collegandolo ad altri servizi e le funzioni di base sono comunque garantite senza fare il minimo cambiamento

L'architettura a Microservizi non ha una sua definizione generale, se non quella di variante del Service-Oriented Architecture, ma è definita da "collezioni di definizioni": è un'architettura in cui un'applicazione è formata da una collezione di "servizi" lascamente accoppiati, di granularità fine (ossia con un obiettivo limitato e specifico) e che usa protocolli di comunicazione leggeri (10).

I servizi in particolare sono caratterizzati come:

- Processi che comunicano attraverso la rete per realizzare un obiettivo usando protocolli technology-agnostic.
- Organizzati attorno alle capacità di business.
- Sono implementati usando differenti linguaggi, basi di dati, hardware e ambienti software, in base alle migliori condizioni.
- Piccoli, attivabili da messaggi, legati ad un contesto, sviluppati autonomamente, distribuiti in maniera indipendente, decentralizzati, costruiti e rilasciati tramite processi automatici.

Quindi non si sta parlando più di un'architettura a strati fortemente dipendenti, ma più di "pezzi di funzionalità" autocontenuti con interfacce chiare e che possono lavorare da soli o diventare parte di una struttura a strati, sempre con caratteristiche di indipendenza e slegabili in qualsiasi momento.

Il metodo con cui questi servizi possono contribuire ad una funzionalità richiesta per l'utente, è dato dalle interfacce di comunicazione precedentemente menzionate: tutti i servizi ne hanno una e questa permette di comunicare, richiedendo o effettuando operazioni tramite il servizio stesso. Queste interfacce devono allo stesso modo incarnare i concetti precedentemente esposti. Spesso, si utilizzano tecnologie, di base, come API REST, che permettono di usare una tecnologia relativamente semplice e a massima diffusione come HTTP, per poi personalizzare la comunicazione in base alle necessità applicative. In questo modo, basta impostare l'interfaccia per far interagire servizi che potrebbero essere scritti anche con componenti diversi.

Parlando poi dell'architettura di un sistema a Microservizi, risulta intuitivo l'utilizzo di tecnologie come il Cloud e Container, che permettono ancor più agilità nello sviluppo, nel deployment e nella manutenzione di questo nuovo paradigma di sviluppo per applicazioni distribuite.

2.5. Containerizzazione – OS-Level Virtualization

Come detto nel paragrafo precedente, l'architettura a Microservizi funziona grazie allo scambio di messaggi tra servizi attraverso l'interfaccia di comunicazione, ossia le API, al paradigma a container e all'orchestrazione. Gli ultimi due in particolare giocano un ruolo fondamentale nella flessibilità e adattabilità dei microservizi.

Per comprendere i container, dobbiamo prima riferirci al paradigma di virtualizzazione definito come "Virtualizzazione OS-Level" o "Software containers": il principio è quello di separare l'applicazione dal sistema operativo e dall'infrastruttura fisica sottostante che si connette alla rete. Il principio è lo stesso, in un certo senso, dei "Sandbox", che sono dei contesti applicativi limitati, con risorse limitate e permessi limitati, che permettono di eseguire un programma qualsiasi e tenerlo sotto controllo, senza che esso possa influenzare il sistema in cui viene lanciato. È un esempio del concetto di software containers, tant'è che Windows utilizza una tecnologia di sandboxing chiamata "AppContainer" (11).

Un primo esempio storico di software container è "Chroot jail": era una best practice per i sistemi Unix-like, che consisteva nell'utilizzo di "chroot" (comando "change root") per cambiare la directory su cui un processo può lavorare e lo stesso accade ai possibili processi figli che vengono creati dal parent. Purtroppo "era" una best practice, perché si sono rivelate parecchie vulnerabilità e perché se il processo opera con privilegi di root, chroot è completamente inutile. Inoltre, non permette di gestire le risorse allocate al processo.

2.6. Containerizzazione - Linux Kernel Containment e Docker

Dopo "chroot", vennero i server virtualizzati, che usavano la normale definizione di virtualizzazione: ogni server virtuale equivaleva ad un OS dedicato e l'applicazione, ma non si tratta ancora di OS-Level Virtualization, dato che i server virtuali sono comunque VM (quindi abbiamo Hypervisor di tipo 1 o 2, OS dedicato e applicazione).

Linux Kernel Containment è il primo esempio di virtualizzazione moderna, per quanto concerne la software containerization. LXC sfrutta i cgroups (una feature del kernel che permette di gestire e controllare le risorse per una collezione di processi) per isolare i processi dalle risorse fisiche e di rete. In più, l'uso dei namespace permette di isolare l'applicazione dal sistema operativo e separare l'albero di processi, l'accesso alla rete, gli ID utente e l'accesso ai files. Si può considerare come una soluzione intermedia tra "chroot" e una macchina virtuale.

Il vantaggio dei container linux sta proprio nel fatto che permettono all'amministratore di virtualizzare solo l'applicazione, senza dover virtualizzare una "macchina intera" ad hoc per far girare l'applicazione. Ciò comporta che non siano più necessarie licenze per sistemi operativi e per altri software che andrebbero normalmente inseriti nella macchina virtuale oltre all'applicazione normale. In termini di overhead, i container sono estremamente leggeri, in quanto l'applicazione usa le normali chiamate di sistema e interfacce di comunicazione senza dover emulare il tutto. E' notevole poter virtualizzare solo l'applicazione proprio perché permette di evitare sforzi computazionali ulteriori e risorse "eccessive", come per esempio un'intera macchina virtuale. Se ci si pensa, per una sola applicazione bisognerebbe avere un sistema operativo e un interprete che emula le chiamate di sistema per il sistema sottostante e le ritraduce per inoltrarle all'interno della macchina. Per una sola applicazione, lo sforzo è immane, se si considera che, nei server virtualizzati, su una sola macchina si hanno più macchine virtuali che rappresentano i singoli server virtuali.

Docker, ad esempio, è un set di prodotti "Platform-as-a-Service" che usa la virtualizzazione OS-Level per lo sviluppo, in cui il software è contenuto in pacchetti chiamati "containers". La definizione di container in Docker, è leggermente diversa da quanto già visto, ma incarna completamente l'obiettivo della virtualizzazione software. I container sono "un'unità standardizzata di software": contiene in un'unica soluzione codice, dipendenze, strumenti di sistema e runtime, librerie di sistema e impostazioni, per fare in modo che una volta installata, l'applicazione possa eseguire senza alcun problema, in un ambiente computazionale distinto e isolato dal sistema stesso e dagli altri container.

In particolare, una volta creato il pacchetto, si ha un'Immagine Docker Container. Nel momento in cui viene caricata sul Docker Engine, questa diventa un container a tutti gli effetti (12) e l'Applicazione può funzionare tranquillamente, appoggiandosi al kernel del sistema che ha installato l'Engine, mentre invece, una macchina virtuale avrebbe il proprio sistema operativo, quindi il proprio kernel.

Poi, in base al tipo di Hypervisor, o si ha una traduzione rispetto alla macchina stessa delle chiamate effettuate dalla macchina virtuale, oppure si ha una doppia traduzione: prima viene chiamato il kernel dell'OS virtuale, poi le chiamate vengono tradotte per l'OS sottostante, per arrivare al kernel dell'OS sottostante e tornare indietro.

Al centro di questo sistema, poi, c'è il Docker Engine, il livello intermedio posto tra i container e il sistema operativo. Il Docker Engine permette di effettuare il deployment e la gestione dei container.

Per questo dispone di due interfacce:

- **Docker CLI**: una Command Line Interface per la gestione e l'amministrazione dei containers.
- **Docker API**: una interfaccia di comunicazione che permette ai container di comunicare sia con l'engine, che tra containers. Non solo permette la comunicazione, ma tramite API si può registrare un Container al servizio di discovery, che permette ai containers di individuarsi tra loro senza necessità di conoscere i dettagli sottostanti.

Docker in particolare, per come funziona, si sposa molto bene con i microservizi. Non solo l'operazione di build (creazione dei container) è già estremamente rapida di per sé, contenendo tutte le dipende e le librerie necessarie all'applicativo per funzionare, ma l'utilizzo del paradigma architetturale SOA risulta in container molto più agili: scomponendo la propria applicazione in microservizi, questi possono essere implementati come app containerizzate. In questo modo, la build è ancora più semplice, data la leggerezza del container stesso, a livelli di contenuto del container e di complessità, con molte meno dipendenze da risolvere e librerie da inserire, ma li rende ancora più rapidi nelle operazioni di manutenzione, come l'interruzione, distruzione, rebuild e rimpiazzo dei container.

Solitamente, per costruire un container attraverso uno dei servizi del Docker Engine, il Docker Builder, si utilizzano i DockerFiles. Questi permettono di specificare come è formato il container e il suo contenuto e come tale viene letto e il builder procede a costruire il container. Se dovessimo usare l'approccio monolitico, il dockerfile sarebbe di dimensioni considerevoli e le operazioni di distruzione, rebuild e rimpiazzo richiederebbero un tempo comunque alto. Invece, disaccoppiando le componenti secondo un approccio a microservizi, si possono ottenere immagini leggere che comunicano sfruttando il sottosistema di networking di Docker (nel Docker Engine).

Il ragionamento è quello, nell'ambito dei Microservizi e della containerizzazione, di:

- Un servizio, un container.
- Servizi intercomunicanti.
- Più immagini di uno stesso servizio disponibili: inteso sia come "Immagini Docker", in modo che se un container ha qualche problema, si sostituisce con una immagine di backup ed è immediatamente disponibile, sia come replicazione, con più container che contengono lo stesso servizio.

Riassumendo, abbiamo quindi:

- Un'applicazione formata da componenti staccate tra loro, i servizi.
- Questi sono posti anche su macchine diverse, che comunicano tra loro tramite sistemi di comunicazione e interfacce comuni, in genere HTTP-based.
- Un servizio, su una macchina, è contenuto in un container (nel qui caso specifico, Docker). Questi sono delle specie di Sandbox in cui l'app opera, in maniera separata e distinta dal resto del sistema e dei container, ma che ha a disposizione, tramite un componente intermedio (il Docker Engine), che fornisce API per interfacciarsi con la rete esterna ed interna (verso altri containers).

2.7. Comunicazione service-to-service

Da quanto visto nella struttura dei Microservizi, i componenti comunicano tra loro attraverso la rete in maniera remota, esponendo delle interfacce di comunicazione con cui chiedere operazioni ad ogni servizio.

Il come è già stato preso in considerazione inizialmente, imponendo alla base del modello di interfacce le API, possibilmente REST-like. Queste interfacce, come detto, permettono di interrogare i servizi sfruttando la modalità con cui HTTP comunica: le operazioni HTTP Get, Put, Post e Delete vengono modificate per indirizzare determinate risorse. In un servizio, queste risorse sono le funzionalità del servizio stesso. E' un sistema molto flessibile, che permette di rendere omogeneo il sistema con cui interrogare i servizi, rendendoli ancor più technology-agnostic. Questo perché il protocollo HTTP è la base con cui vengono create le interfacce ed essendo un protocollo strutturale con una presenza costante nel mondo web, è utilizzato in qualsiasi ambito dello stesso, quindi accettabile da qualsiasi entità che si affacci su una rete web.

Se il come identifica l'aspetto con cui un servizio si interfaccia con l'esterno, un altro aspetto importante è il dove comunicare, ossia il canale di comunicazione. Ci sono molti modelli per il canale di comunicazione in una struttura a Microservizi, alcuni mutuati dall'ambiente distribuito e altri proveniente dall'ambiente SOA. E' importante sapere quindi la struttura su cui comunicano i servizi, perché, sempre indicando la struttura a Microservizi, può essere di vitale importanza per l'organizzazione di strategie difensive.

2.7.1. Socket, connessioni point-to-point e Discovery

Normalmente, due macchine in rete comunicano indicando dei punti d'accesso su cui il programma si mette in ascolto per ricevere e rispondere a messaggi, ossia le socket. Quindi avremmo una connessione diretta su socket, con uno scambio di messaggi diretto.

Pura e semplice connessione end-to-end. Due servizi si metterebbero in comunicazione utilizzando un canale di comunicazione diretto su cui scambiarsi i messaggi. In genere, per rendere più sicuri i canali, si utilizzano altri protocolli e sistemi, come ad esempio TLS o IPSec, che permettono di rendere la comunicazione sicura e garantire prestazioni di sicurezza elevate. Il canale è semplice e diretto, ma, in un ambiente a Microservizi in cui

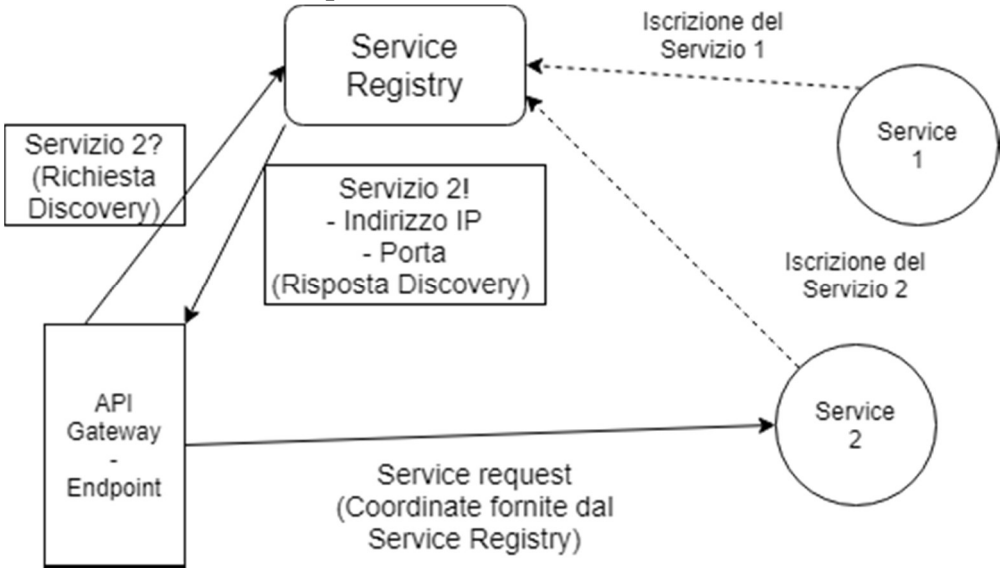

Figura 4 – In una rete con sistema di discovery, quando si vuole contattare un servizio, si chiedono al Service Registry i riferimenti per contattarlo. Il Service Registry verifica se almeno uno dei servizi registrati è quello richiesto (ci possono essere anche più repliche di un servizio) e fornisce le indicazioni verso il primo servizio disponibile.

la struttura può cambiare rapidamente, c'è bisogno di un metodo per indirizzare correttamente i servizi verso le loro controparti necessarie. In genere si può o definire una lista di indirizzi statica a cui ogni servizio può fare riferimento, oppure usare un servizio di Discovery. Il servizio di discovery funziona da "instradatore": i servizi si registrano al registro del servizio di discovery per essere "visibili" e, quando un entità cerca un determinato servizio, il servizio di discovery passa le coordinate del servizio d'interesse, in modo che i tramiti instaurino una connessione diretta. Lo sfruttamento di un servizio di discovery è molto utilizzato nell'ambito dei Microservizi. L'uso dei container rende mutevole la struttura interna della rete, con container che vengono cambiati continuamente e punti d'accesso che cambiano indirizzo. Il servizio di discovery evita di utilizzare indirizzi statici scalando in maniera più ottimale, permettendo di inserire e togliere host e servizi senza creare problemi di struttura di rete.

2.7.2. Enterprise Service Bus

Un pattern "architetturale", per cui tutte le comunicazioni passano attraverso un unico "bus" implementato generalmente tramite middleware. La sua flessibilità, al netto degli svantaggi, lo individua come pattern fondamentale nelle architetture SOA e Microservizi, anche se, per il secondo caso, limita il concetto di decentralizzazione e disaccoppiamento. Ogni componente del sistema, che sia un'applicazione od un servizio, si affaccia tramite un'interfaccia sul bus e l'ESB provvede ad effettuare la consegna al componente di destinazione. La potenza dell'ESB risiede in tre concetti principali: la centralizzazione della comunicazione, la capacità di connettere servizi con comunicazioni technology-agnostic e technology aware e la possibilità di implementare diversi modelli di comunicazione.

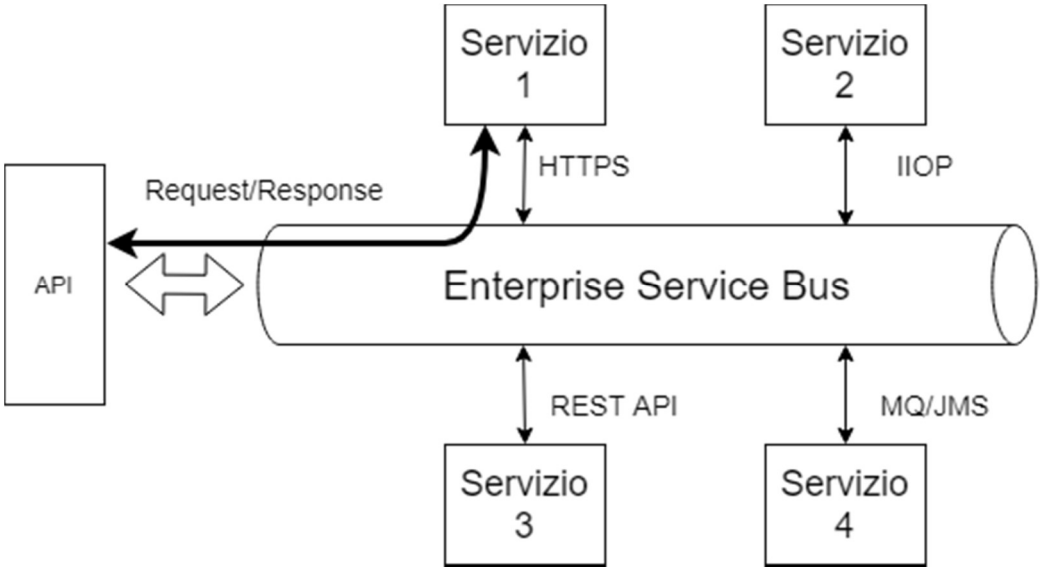

Figura 5 – Dall'immagine si può notare come l'ESB è il canale comune su cui viaggiano tutte le richieste nel sistema. Inoltre, non importa la tecnologia con cui il servizio comunica, l'ESB si fa carico della traduzione.

La centralizzazione della comunicazione è un aspetto estremamente comodo: si ha un unico componente in cui passano tutte le comunicazioni e che è progettato per amministrarle sotto ogni punto di vista: routing, mediazione, orchestrazione, gestione avanzata di sicurezza e amministrazione di sistema e modelli di sicurezza standardizzati. La flessibilità dell'ESB consiste nella capacità di astrarre molte componenti e di renderle facilmente integrabili, che sia l'aggiunta di un nuovo servizio, la gestione di quelli esistenti o l'implementazione di meccanismi di sicurezza o policy unificate. L'altra faccia della medaglia, in questo caso, è che è tutto centralizzato e rappresenta un punto di fallimento singolo, una fallacia intrinseca della centralizzazione.

L'obiettivo dell'ESB in termini di comunicazione è quello di tradurre i messaggi per ogni servizio. Indipendentemente dal protocollo utilizzato, ESB prende il messaggio e lo consegna tradotto per il protocollo corretto del destinatario, ad esempio da SOAP/HTTP a JSON/HTTP e viceversa. Sebbene non sia da abusarsi, in quanto la traduzione richiede

comunque risorse computazionali non trascurabili, permette di avere una flessibilità tale da ammettere l'uso di diverse tecnologie che convivono in uno stesso sistema. Tuttavia, la diversità tecnologica porta con sé anche una maggiore complessità nel gestire quelle che possono essere le vulnerabilità di tali modelli di comunicazione: anche se l'ESB contiene tutta la logica di controllo, protocolli diversi possono avere vulnerabilità diverse che possono combinarsi tra loro, come anche un'implementazione errata delle interfacce di comunicazione può rappresentare una vulnerabilità.

L'idea principale dietro l'ESB è quello di utilizzare un modello di comunicazione principalmente basato su eventi e a scambio di messaggi. Ciò significa che la comunicazione può essere punto a punto, con opportune code di messaggi per ogni servizio affacciato sul bus, oppure può essere impostata con il modello publisher-subscriber. E' il modello comunicativo per l'ESB più diffuso e che genera più confusione, in quanto spesso non è facile distinguere se l'ESB sia un semplice Message Broker o meno, nonostante sia concezione diffusa che il pattern Message Broker sia una parte dell'ESB (13). Ciò che richiede attenzione, per questo aspetto dell'ESB, è gestione della congestione del traffico tra servizi e il controllo dei messaggi, ma spesso gli ESB, commerciali e non, sono forniti di strumenti per la gestione del traffico e del bilanciamento di carico.

2.7.3. Modello Publisher-Subscriber

Abbiamo visto che l'ESB, in sostanza, fornisce funzionalità simili al modello Publisher-Subscriber puro, con l'implementazione di molti altri accorgimenti a contorno per renderlo funzionale e trasversale. Tuttavia, anche un semplice modello pub-sub è utilizzabile come meccanismo di comunicazione. L'implementazione effettiva tramite code o topic è irrilevante, ciò che è più interessante è la scelta di avere uno o più Message Broker.

Il modello publisher-subscriber funziona con questo componente: i servizi si registrerebbero ai Message Broker come publisher o subscriber (o entrambi), per poi "produrre" messaggi in base alle richieste dei subscriber. Il sistema diventa macchinoso, certamente, nel momento in cui dobbiamo stabilire come i Message Broker debbano funzionare e quindi quanti ne servono. Se dovessimo applicare il modello pub-sub nella comunicazione tra servizi di MicroBank, dovremmo scegliere se avere un unico Message Broker, o se averne almeno due. In sistemi più complessi, con un numero di servizi più grande, il numero di Message Broker crescerebbe di pari passi con il numeri di servizi e aumenterebbe la difficoltà nel gestire ogni singolo Message Broker. Per capire meglio, bisognerebbe gestire l'insieme di regole per cui si distingue "chi" può pubblicare o sottoscriversi "a quale topic/coda" e "quali informazioni" possono essere prodotte. Per non parlare della QoS da negoziare, nel caso dei topics.

In sintesi, ci possono essere diverse modalità con cui i servizi possono comunicare tra loro. Tali modalità portano con sé problematiche che, a livello di amministrazione e sicurezza, vanno gestite in maniera adeguata.

2.8. Autenticazione e autorizzazione user-to-service e service-to-service

Come in ogni applicazione distribuita, modellata come Web application, uno dei discorsi fondamentali è l'autenticazione e, di riflesso, l'autorizzazione degli utenti. Il discorso di autenticazione e autorizzazione è comune sia per l'architettura monolitica che per l'architettura Microservizi in quanto, per definizione, ha degli obiettivi sostanziali:

- **Autenticazione**: confermare l'identità (utente) della persona che sta accedendo a determinati servizi.
- **Autorizzazione**: definire le modalità di accesso ai servizi facenti parte del sistema, in base alla loro identità (sempre inteso come utente).

La letteratura in merito è vasta a sufficienza e non è necessario addentrarsi nel dettaglio nell'argomento dell'autenticazione e dell'autorizzazione. Ciò che è di interesse, è l'insieme degli strumenti di autenticazione e come essi funzionano. L'architettura monolitica, attualmente, utilizza un sistema di autenticazione basato su fattori multipli e sfrutta il concetto di sessione per fare in modo che l'utente sia sempre autenticato e autorizzato quando accede al servizio. Questo perché nell'architettura monolitica tutti i componenti sono, come abbiamo già detto, all'interno della stessa "applicazione" e quindi le informazioni di sessione sono a disposizione di tutta l'applicazione e possono anche essere passate tra le istanze dell'applicazione stessa, su un server.

2.8.1. Metodi di autenticazione

In prima istanza, bisogna associare l'utente che contatta il servizio con il proprio "account", la propria identità all'interno di quel servizio e che contiene tutte le informazioni circa l'utilizzo del servizio stesso da parte dello specifico utente. A questo proposito, bisogna riconoscere quindi un "utente" e collegarlo al relativo account in maniera sicura. Per identificare un utente, si riferisce alla teoria dei "fattori" (14): per autenticare una persona ci si basa essenzialmente su.

- **Qualcosa che si sa**: un'informazione nota solo ad entrambi permette di autenticarsi tra pari. Un esempio sono i segreti, in termini informatici le password (parola d'ordine, dunque segreta e nota solo ai due pari). Uno dei metodi più utilizzati
- **Qualcosa che si ha**: un oggetto che permette di autenticarsi. Il fatto di possedere l'oggetto e di dimostrarne il possesso è la "prova" che autentica l'individuo. Un esempio sono schede di sicurezza, badge/tesserini, oggetti che posseggono informazioni uniche (come i token di sicurezza, che forniscono PIN di riconoscimento).
- **Qualcosa che si è**: caratteristiche fisiche. I dati biometrici, l'impronta digitale o la retina sono i più noti. In ambito informatico, alcuni sistemi utilizzano il riconoscimento vocale o il riconoscimento facciale.

Nell'ambito umano, in genere, anche se si utilizza solo uno di questi fattori, in realtà altri fattori entrano in gioco, che permettono una persona di effettuare una specie di "autenticazione critica" dell'individuo di fronte a sé. Le macchine invece non possono farlo. Possono autenticare la "macchina" che comunica con loro, ma non possono identificare chi ci sia dietro la stessa, quindi bisogna inventare metodi che permettano di autenticare "inequivocabilmente" l'utente che si trova dietro la macchina. A partire dalla definizione di fattori, dunque, si ottengono due famiglie di metodi di autenticazione.

2.8.1.1. Single-Factor Authentication (SFA)

Per autenticare un utente si utilizza un solo fattore tra quelli precedentemente elencati. Nell'ambito delle applicazioni e servizi Web e distribuiti, è stato il più comune, in quanto molto facile da implementare, molto leggero in termini di memoria e molto semplice in termini di informazioni scambiate. In generale, il maggior rappresentante di questa categoria è **Password-based authentication.** L'utente viene autenticato a mezzo di determinate informazioni, ossia le credenziali d'accesso, tra cui è presente una password. Inserire dunque le credenziali consente di autenticare l'utente presso il servizio. Quindi utilizza un solo fattore, che si basa su delle informazioni "segrete" (cioè "qualcosa che sai"). Questo metodo è ovviamente limitato in più di un aspetto e la letteratura sul furto o sull'individuazione di queste informazioni è estremamente ampia.

Non è detto che sia un sistema di autenticazione debole, ma non è reputato un sistema sufficientemente forte attualmente. Si possono usare tecniche che permettono di proteggere questi segreti e di complicare qualsiasi operazione che miri a carpire tali segreti (crittografia, limiti su lunghezza e alfabeto di generazione delle password et simila), tuttavia il maggiore rischio è sempre lo stesso: avere le credenziali di autentica completamente come il possessore di tali credenziali, quindi come l'utente stesso.

C'è un altro fattore che rende la Password-based authentication pericolosa, come SFA: secondo un sondaggio effettuato da Google/Harris nel 2019 (15), circa il 65% del campione riutilizzava le stesse password. In caso di attacco informatico e conseguente data breach di credenziali, su servizi magari minori o meno protetti, queste credenziali potevano essere usate per violare account su servizi maggiori, come ad esempio un account Microsoft (16). Ciò significa creare una reazione a catena a partire da una sola fuoriuscita di credenziali.

2.8.1.2. Multi-Factor Authentication (MFA)

La debolezza intrinseca di avere un solo "fattore" di autenticazione venne individuata come un problema già molto tempo fa. Per questo, si è deciso di passare a metodi che non utilizzassero un solo "fattore" di autenticazione, ma più fattori contemporaneamente o in combinazione l'uno con l'altro. I primi tentativi di utilizzare servizi che richiedessero più di un fattore di autenticazione, sono datati già nel 2005 (17). Il concetto di "Internet banking", cioè di avere servizi bancari online, ha sollevato la necessità, a causa del crescente numero di reati informatici di furto di credenziali e frodi, di cercare metodi di autenticazione più forti, oltre ad ulteriori linee guida che potenziassero la sicurezza e la stabilità nell'ambiente dell'"Internet Banking". Successivamente, molti metodi hanno concorso ad arricchire l'insieme di soluzioni multi-factor, in particolar modo Two-Factor Authentication (2FA):

- Utilizzo di SMS per passare ulteriori passcode o password monouso temporanee (OTP). Nonostante sia ancora il più diffuso, nell'ambito 2FA, non è considerato molto sicuro a causa della poca sicurezza del sistema di comunicazione telefonico, tanto da portare il NIST nel 2016 a deprecarlo come fattore di autenticazione e ad imporre che, se utilizzato come autenticatore, il dispositivo ricevente debba essere identificato univocamente (18) (19) (20)
- Tra il 2016 e il 2017, Google ed Apple hanno sostituito questo sistema con un sistema di notifiche push, sfruttando quindi la comunicazione via internet dei sistemi mobili e delle applicazioni mobili in sostituzione della comunicazione via SMS, rendendo lo "smartphone" un metodo come forte come fattore di autenticazione per "ciò che si ha" (21) (22).

Il trend negli ultimi anni è stato quello di passare a più di 2 fattori di autenticazione.

Molti servizi bancari, ad esempio, in precedenza utilizzavano due sistemi di autenticazione, per quanto riguardava l'internet banking: sia le credenziali utente, che un token che generava codici monouso temporanei, che venivano usati come ulteriore misura di sicurezza.

Nel 2015 entrò in vigore la Revised Payment Service Directive (23) (PSD2), in cui viene utilizzano il concetto di Strong Customer Authentication (SCA), per cui sono necessari almeno 2 o più fattori che non siano collegati l'uno con l'altro per autenticare un utente, in modo che la violazione di uno di questi fattori non comprometta l'affidabilità dei fattori non violati. Le direttive circa la SCA sono state rese effettive dal 2019, con tolleranza di adattamento a tali direttive in merito per fine 2020. Infatti, molti servizi bancari hanno dismesso sistemi come il token, in favore di applicazioni che sfruttano sia un sistema a token che genera codici monouso temporanei (24), sia ulteriori dispositivi presenti negli attuali smartphone di più comune uso, come l'impronta digitale, per implementare ulteriori "fattori" di autenticazione.

Purtroppo, al di là di queste realtà per cui la sicurezza degli utenti è una missione critica, il modello più diffuso di autenticazione multi-fattore utilizza solo due fattori (in particolari, comunicazioni telefoniche tramite SMS o notifiche push tramite app smartphone, in coppia con le credenziali utente).

2.8.2. Autenticazione e autorizzazione persistente – Metodi Stateful e Stateless

L'interazione con un servizio web implica di aprire una connessione, eseguire determinate operazioni, ottenere i risultati e chiudere la connessione. Tale interazione, spesso, non è solo "atomica": prendiamo l'esempio classico dell'e-commerce e vediamo che, accedendo al servizio web, abbiamo bisogno di mantenere determinate informazioni collegate a noi utenti. Non solo, durante la sessione, il servizio deve essere in grado di mantenerci autenticati, senza avere necessità di ri-autenticarci ad ogni operazione. Ciò significa che bisognerà mantenere "memorizzato" da qualche parte il fatto che l'utente si sia autenticato e tutta l'applicazione web deve saperlo. Questo è dovuto al fatto che HTTP, di base, è "stateless", cioè non può essere utilizzato per ricordare determinati dettagli appartenenti al contenuto della comunicazione. La risoluzione consiste in utilizzare metodi che permettano di associare l'autenticazione (e quindi l'utente) a determinate informazioni che consentano di dire "questo utente si è già autenticato, pertanto può procedere all'utilizzo del servizio", almeno finché l'utente non chiude la connessione.

2.8.2.1. Session-based Authentication – Stateful e Distribuita

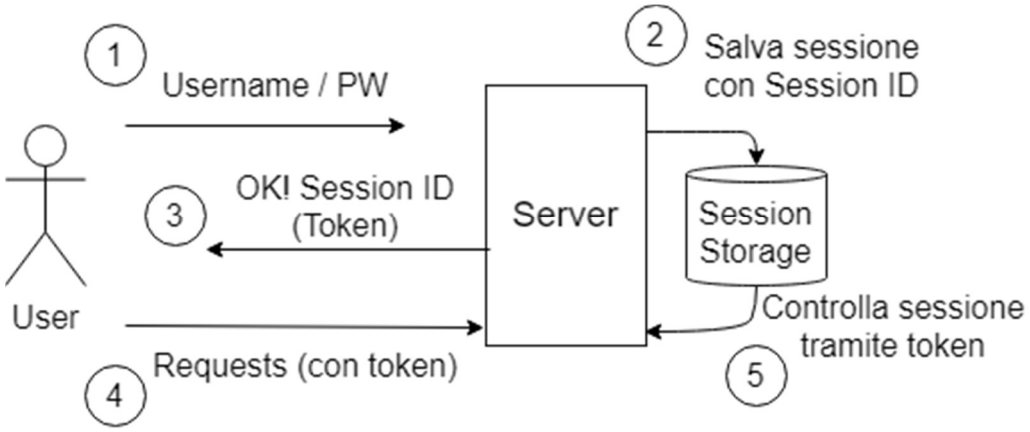

Figura 6 – Uno scambio di autenticazione, con i passaggi numerati. La Session-based Authentication si nota dal fatto che la sessione viene salvata (2) e viene passato un token (3) da accompagnare ad ogni richiesta (4), che viene controllato per verificare l'identità (5).

Una volta eseguito il login, all'utente viene dato un identificativo, che permette al Server di riconoscerlo. Questo identificativo viene messo dentro un "token" (un Cookie) che permette di "salvare" l'identificatore lato utente. Lato server, invece, il server salva i dati relativi all'autenticazione e alla sessione dell'utente e vi associa l'identificativo da dare indietro all'utente nel Cookie (appunto, il "session_id"), insieme ad altre informazioni. Il Cookie, infine ha una data di scadenza per evitare che il cliente sia "perennemente" autenticato e anche i dati di sessione possono scadere, necessitando che l'utente si ri-autentichi. In questo modo, l'utente è sempre autenticato presso il server, inviando il cookie ad ogni richiesta, fino a quanto la connessione non viene chiusa. Ovviamente è il server che si fa carico di mantenere il collegamento tra l'identificativo di sessione (o Session ID) e il record dell'account utente. Una volta scaduto il cookie, è necessario ri-autenticarsi presso il server per ricevere un nuovo session ID.

Questo approccio rende il server "stateful", cioè lo stato di sessione viene salvato e viene dato un riferimento a quei dati al client, che lo passa al server per "rintracciare" i dati di sessione in memoria. E' uno dei sistemi più utilizzati, che è stato associato ai "Cookie" in quanto sono i contenitori di quelle informazioni, in particolar modo il Session ID, che permettono quanto detto precedentemente. E' molto comodo, nell'attuale mondo web, perché nell'architettura monolitica, i servizi dell'applicazione possono verificare che l'utente sia autenticato, oppure gestire l'autenticazione in maniera centralizzata.

Nonostante sia stato un sistema molto sicuro per la gestione dell'autenticazione di sessione, non è esente da vulnerabilità: il cookie può essere intercettato ed utilizzato per autenticare automaticamente un terzo individuo, che "impersonerà" l'utente oppure può essere fatto creare all'utente di proposito per essere ceduto all'attaccante, come negli attacchi CSRF. Ciò che rende preoccupante tale vulnerabilità è la natura e durata del cookie stesso: il cookie permette di non inserire le proprie credenziali ogni volta per ogni

operazione, ma sotto la superficie, il cookie viene usato come "credenziale" ogni volta che viene associato al record di sessione salvato sul server. Per questo può autenticare un utente in maniera persistente. Se avesse una durata "illimitata", chiunque fosse in possesso del cookie, potrebbe impersonare l'utente finché o non viene invalidata la sessione (per cui si ricomincia da capo la comunicazione e quindi si riparte dall'autenticazione), oppure quando viene invalidato il cookie stesso (ad esempio, si elimina il record di sessione del relativo utente). Questo ha portato al fatto, in determinati casi, di rendere il cookie poco durevole: in questo modo, la durata della violazione coinciderebbe con la durata del cookie stesso, diminuendo di molto la finestra temporale di un possibile attaccante per effettuare operazioni complesse. Sono state sviluppate anche tecniche per prevenire determinati attacchi ai cookies, come i flag. Ad esempio, "HttpOnly" impedisce la lettura tramite Javascript client-side, "Secure" permette di validarli solo se la connessione viaggia su HTTPS o flag di tipo X-CSRF che permettono di intervenire a protezione di attacchi CSRF.

L'unico inconveniente maggiore di questo tipo di autenticazione è che le performance degradano quando si vuole scalare la Session-based Authentication lateralmente e questo problema è molto importante nell'ambito Microservizi: lo stato di sessione è mantenuto dal web server, ma se ci fossero repliche, questo stato deve essere condiviso anche con le altre repliche, per fare in modo che il cookie sia valido per tutti i web server, non solo per il primo che autentica e accetta la connessione dell'utente. Infatti questo problema relativo alla sessione, a cui sono già state trovate soluzioni, ha portato al concetto di Distributed Session, in Figura 7, che risolve il problema di avere stati di sessione locali e multipli.

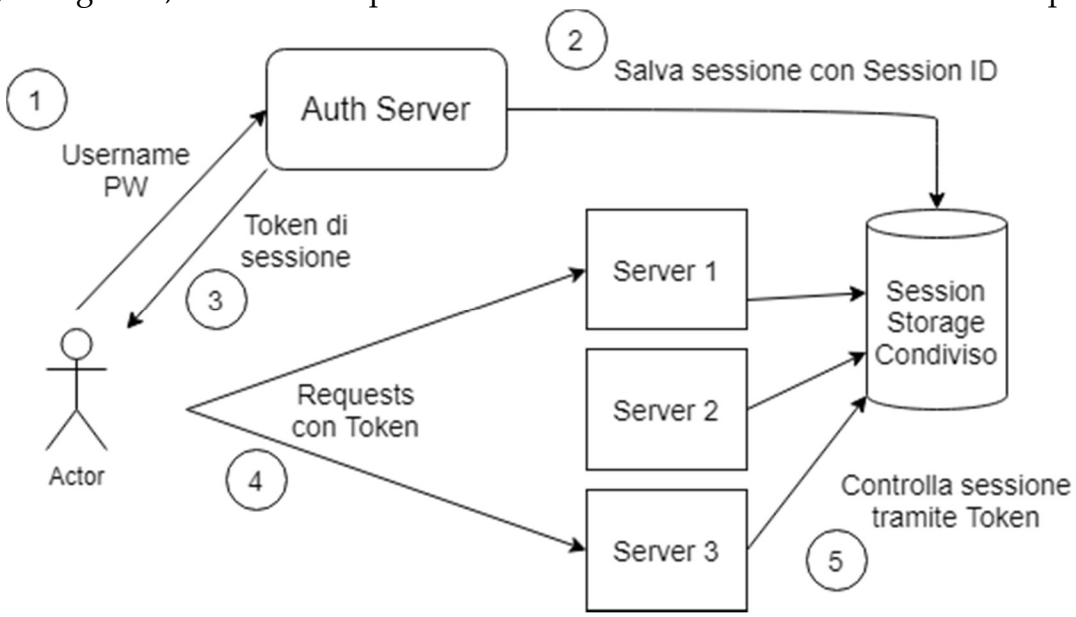

Figura 7 – Contrariamente alla Session-based normale, qui la struttura di stoccaggio dei dati di sessione è una struttura esterna, a cui tutti i server accedono. In questo caso, un Auth Server è predisposto per l'autenticazione, ma anche un singolo server può incaricarsi dell'autenticazione. L'importante è che non salvi i dati localmente, ma li salvi nello storage condiviso.

2.8.2.2. Token-based authentication (JWT) – Autenticazione Stateless

Simile alla versione Session-based, ma con l'obiettivo di "evitare" di dover salvare i dati di sessione sul server. Per riuscire nell'intento, quando un utente si autentica tramite l'utilizzo delle credenziali, il server crea un "token" temporaneo che contiene i dati utente. Questo viene inserito nell'header di risposta all'utente e l'utente lo salva dal proprio lato. Quando l'utente invia richieste, inserisce il token nell'header e il server lo verifica. Quando l'utente chiude la connessione, il token viene eliminato lato client.

Questo fa in modo di rendere il server "stateless". Mentre nella Session-based l'identificatore va collegato ai dati di sessione stoccati ogni volta che si riceve, il token invece viene solo generato (e firmato) dal server e viene inviato al client, che si occupa di memorizzarlo. In questo modo, qualsiasi server a cui viene presentato, lo verifica e i dati contenuti nel token autenticano in automatico l'utente.

Il vantaggio, contrapposto a Session-based, è che scala molto bene lateralmente. Il token può essere presentato a qualsiasi server facente parte del servizio e questo verrebbe riconosciuto, autenticando in automatico l'utente a prescindere dal server contattato (purché il server sappia come leggerlo). Senza quindi l'onere di passare i dati di sessione tra i vari server, l'autenticazione diventa molto più agevole in termini di servizio.

Per capire come è fatto un token, si può fare riferimento ad uno standard per questo tipo di autenticazione, cioè JWT o JSON Web Tokens. JWT è uno standard aperto (25) che trasforma oggetti JSON in una forma compatta per trasmetterli in maniera sicura. In particolare, la struttura della forma compatta di un JWT è strutturata come tre campi, separati da punti:

- **Header**: contiene informazioni sull'algoritmo di firma del token e sul tipo di token. Per JWT ovviamente il tipo è JWT.
- **Payload**: contiene i dati significativi del token.
- **Signature**: la firma digitale del token.
- **Struttura del JWT**: <header>.<payload>.<signature>

A sua volta il Payload si compone in dati, detti "claims", che si dividono in tre tipologie:

- **Registered claims:** sono delle informazioni standard, non necessarie, ma molto utili per fornire informazioni generali. Alcune claims sono "iss", il generatore del token ("issuer"), "exp", cioè la data di scadenza del token, "sub", ossia il soggetto di destinazione del token ("subject"), e "aud", cioè per quale contesto sia valido questo token ("audience").
- **Public claims**: informazioni di utilizzo generale che possono essere definite da chi utilizza JWT.
- **Private claims**: campi personalizzabili contententi informazioni da condividere con le entità che utilizzano i JWT.

Queste informazioni, nei token JWT sono codificati in Base64Url e sono posti nell'header della richiesta. La motivazione è dovuta alla necessità di trasporto, più che ad una questione di sicurezza, perché la codifica li rende più simili ad una stringa di lunghezza modesta e facilmente incorporabile in un header di richiesta. Sono usati come metodo di autenticazione, inserendo identificativi come "private claims", e vengono inseriti nel campo dell'header "Authorization", inserendo anche lo schema in cui leggerli ("Bearer", di norma).

Di base, sono firmati digitalmente e sono codificati in Base64Url, le informazioni contenute nel token sono leggibili da chi li riceve. Ciò significa che non devono essere usati per contenere segreti (come per esempio "password") a meno di essere anche crittografati, nonostante questo vada a sminuire i vantaggi della caratteristica di semplicità. Come anche, un problema di scorretta implementazione dei Token JWT è quello di contenere dati di sessione, ma ne parleremo successivamente.

Abbiamo detto che il token è mantenuto lato client. In genere, questo è mantenuto in una struttura client, il browser storage o "local storage". Il local storage è una struttura del browser strutturata come "chiave-valore", no-SQL e accessibile tramite JS API, in cui i token possono essere salvati in maniera persistente e permanente (contrariamente al "session storage", che viene ripulito alla chiusura della pagina eliminando anche i token, di conseguenza). Poiché locale, può essere acceduta solo localmente ed in genere viene acceduta dal client per prelevare i token da inserire negli header delle richieste (26).

La semplicità e comodità dei token JWT viene controbilanciata da vulnerabilità che i cookies non hanno. I cookies posseggono una resistenza maggiore ad attacchi di tipo Cross-Site Scripting (XSS) in quanto posso essere protetti dalla lettura (ad esempio tramite l'uso del flag HttpOnly), mentre i token JWT sono completamente leggibili lato client, poiché contenuti nel local storage. Una soluzione sarebbe mantenerli nel "session storage", per fare in modo che la chiusura della connessione con il servizio web cancelli anche i token registrati, di fatto distruggendoli. Però non protegge completamente dal fatto che le informazioni siano lo stesso leggibili, se vi si riesce ad accedere. A questo punto, si può pensare di utilizzare la cifratura simmetrica, per fare in modo di risolvere questo problema.

In realtà, non c'è solo il problema della leggibilità e accessibilità dei token JWT. Un problema di carattere logico appare nel momento in cui, nonostante si sia chiusa la sessione con un servizio web o un sito, il token sia ancora integro: supponiamo che il token non venga eliminato con la pulizia del session storage, ma che sia stato duplicato. Il token è ancora valido, se la scadenza del token stesso non è stata raggiunta. Un problema ulteriore deriva dal fatto che cancellare il token non significa invalidarlo, come anche non esiste un metodo per permettere al client di invalidare il token.

Questo richiede che una forma di invalidazione del token a monte, lato server. Tale misura risulta ancor più importante quando, come abbiamo detto, il token può contenere dati di sessione (come il session_id). Tale forma di invalidazione sarebbe quella di creare una "blacklist" di token: una volta che la connessione viene chiusa con il client, il token fornito dal client viene registrato in una tabella, indicandone concettualmente l'invalidità. Oltre a sconfiggere l'obiettivo "stateless" dei JWT, poiché di fatto si sta mantenendo uno "stato" dei token dell'utente, tralasciando l'uso dei token come contenitori di dati di sessione, questo impone di porre misure di sicurezza per fare in modo che la "blacklist" di token non sia accessibile. Se i token in blacklist fossero accessibili un attaccante potrebbe usarli per autenticarsi, a meno che non siano scaduti. La scadenza permetterebbe di eliminarli a priori, snellendo la tabella, ma il problema rimane. Per questo, sarebbe meglio tenere una "whitelist" contenente i token di sessioni attive e toglierli dalla lista quando le connessioni vengono chiuse, ma il problema di memorizzare uno "stato" dei JWT invalida, di fatto, il concetto di autenticazione "stateless". Inoltre, l'idea di combinare sessione e JWT, come abbiamo visto, complica ancor di più la gestione dell'autenticazione, in quanto i JWT iniziano a diventare molto più simili a dei "certificati" da gestire, ponendo misure di controllo continue sulla loro validità.

Nonostante questo, JWT è comunque un sistema estremamente utilizzato nei Microservizi: il fatto che i servizi sono distribuiti, non centralizzati, significa che l'autenticazione non è possibile a mezzo di sessione, perché dovrebbe essere registrata in ogni microservizio e gestita singolarmente. Invece i token JWT risultano molto più agevoli per trasferire informazioni di contesto tra i microservizi, specie tramite alcune claims: ad esempio, la claim "**aud**" (audience), è conveniente per simulare un sistema di autorizzazione (27). In "aud" si possono specificare i possibili "riconoscitori" di questo token[1], permettendo quindi di specificare nel token a quali servizi può accedere un utente. Questo semplifica di molto la gestione dell'accesso ai servizi, in termini di controllo degli accessi e di sicurezza: un token rifiutato può generare immediatamente un alert, segnalando un possibile abuso del token e permettendo di avere immediatamente nota di una situazione anomala.

Inoltre, JWT non è utilizzato "stand-alone", ma viene usato anche per altri meccanismi più complessi di autenticazione e autorizzazione, come ad esempio OAuth 2.0.

[1] https://tools.ietf.org/html/rfc7519#section-4.1.3. Spiegazione della claim "aud" secondo la RFC stessa.

2.8.2.3. OAuth 2.0 + OpenID Connect (+ JWT) = Single Sign On (SSO)

La domanda, in generale, nell'accesso ad un servizio è "come faccio a sapere chi sei, per farti utilizzare il servizio?". Questo è il concetto di autenticazione (e conseguentemente autorizzazione), nella comunicazione con un servizio Web: "Io ti riconosco e ti permetto di accedere". Nel tempo però, si è presentata la necessità di rendere interoperabili certi servizi web, per facilitarne l'utilizzo. In alcuni casi, un servizio web (ad esempio, la posta elettronica) poteva essere necessario per un altro servizio web. Inoltre, l'avvento del mondo mobile, con la diffusione degli smartphone, necessitava di un sistema di autenticazione più comodo e più facilmente scalabile. Una necessità di questo tipo è emersa nell'ambiente aziendale, con tanti servizi, non tutti appartenenti alla stessa organizzazione (quindi su domini di sicurezza diversi), per cui l'utente doveva essere continuamente autenticato. Quindi dovevi avere due account, in genere collegati tra loro, per usare più servizi contemporaneamente.

In poche parole, avere la possibilità di essere identificati una sola volta per usare più servizi senza doverti ri-autenticare ogni volta per ogni servizio che abbia un gestore diverso (nuovamente, un diverso dominio di sicurezza). Per capirci meglio, oggi, con un account Google, o Facebook, possiamo "avere già un account" su altri servizi, semplicemente usando direttamente l'account Google o Facebook come credenziali. Questo accade perché non "accediamo direttamente al servizio", ma deleghiamo l'accesso all'account Google o Facebook o equivalenti.

OAuth 2.0 [2] è ciò che permette di farlo e di fatto sia Google che Facebook lo utilizzano. OAuth 2.0 è uno standard aperto per l'autorizzazione di servizi web terzi ad accedere ed utilizzare determinate informazioni, delegando l'accesso ad un servizio specifico senza avere bisogno di utilizzare credenziali (28). Significa che, nel caso dell'account Facebook: nell'accesso ad un particolare servizio, chiedo che l'account Facebook autorizzi l'accesso a determinate informazioni da parte del servizio. L'account Facebook richiede il permesso all'utente per far accedere le informazioni dal servizio, fornisce un'autorizzazione in forma di token che può essere utilizzata dal servizio per accedere a queste informazioni e fare in modo che l'utente possa utilizzare il servizio come se avesse un account. Il tutto, senza fornire credenziali, le uniche credenziali fornite sono per autenticarsi presso Facebook stesso.

[2] https://oauth.net/2/, descrizione e specifiche di OAuth 2.0

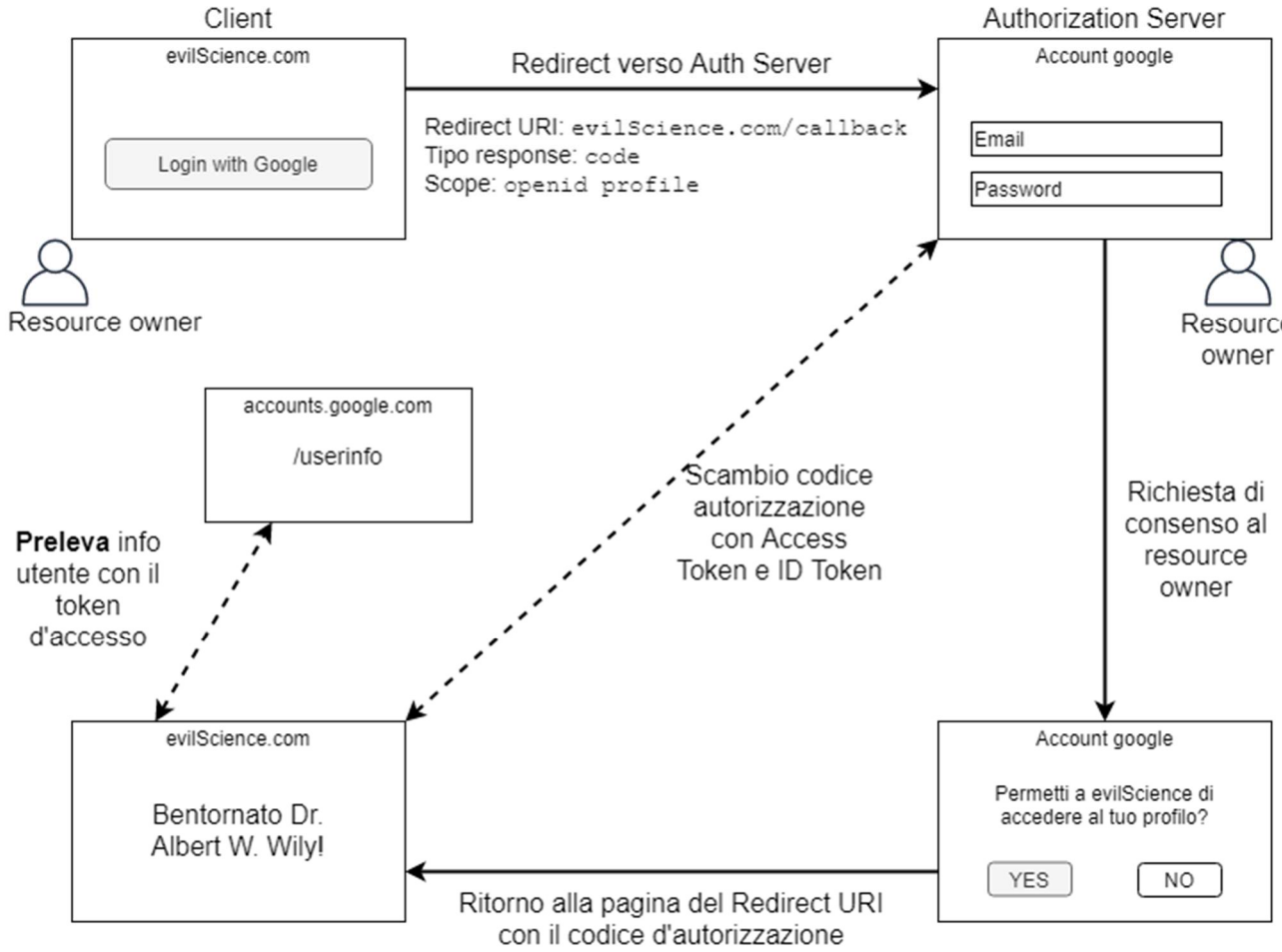

Figura 8 – Questo è il flusso di OAuth 2.0 con OpenID: la presenza di OpenID si nota dalla presenza del dato "Scope", che rappresenta i tipi di dato per cui si rilascia l'autorizzazione al prelievo. Da notare, con l'ID Token, il sito preleva direttamente i dati. Solo con OAuth 2.0, il sito avrebbe invece richiesto i dati ed ottenuto una risposta dall'account di Google. [3]

OAuth 2.0 permette quindi di effettuare l'autorizzazione di un servizio, non "autentica" di fatto un utente. Di fatto, l'account Facebook cede un "token d'accesso", che viene usato per concedere al servizio di accedere a dati contenuti nell'account Google momentaneamente, senza l'uso di credenziali, perché viene dato il permesso dall'utente. Ovviamente non viene dato un accesso completo ed indiscriminato e non viene dato senza un consenso. Una volta che il servizio contatta Google per usare le informazioni dell'account Google, fornisce una lista, chiamata "scope", in cui indica quali informazioni servono e in che modo servono. Ad esempio, uno scope può essere "Accedere alla lista dei contatti in lettura". Questi scope vengono mostrati all'utente, insieme al nome del servizio, quando Google chiede il consenso ("consent") all'utente di cedere queste informazioni a quel servizio.

[3] Immagine presa dalla presentazione "OAuth and OpenID Connect (in plain english)" di Nate Barbettini, Okta.

Quando viene fornita l'autorizzazione al servizio, questo richiede un'"access token", che contiene tutte le informazioni necessarie per far richiedere informazioni al servizio, limitate e circoscritte ai soli "scope" presentati (non può quindi, ad esempio, "accedere al calendario Google", perché non era tra gli scope presentati).

Il doppio passaggio è necessario perché, in accordo con la struttura di OAuth 2.0, la comunicazione avviene su due spazi di trasporto: il "Front channel", considerato sicuro, ma fino ad un certo punto e rappresentato dalla comunicazione tra browser e servizi, e il "Back channel", considerato sicuro a priori. Il "Back channel" è considerato sicuro a priori perché la comunicazione avviene direttamente tra il server del servizio (per cui c'è l'implicita fiducia nel fatto che il codice nel server del servizio sia "sicuro" a priori) e il server che autorizza e/o fornisce le risorse.

La comodità e la "sicurezza" di OAuth 2.0 ne hanno causato la diffusione estrema, con un effetto collaterale: OAuth 2.0 è uno strumento di "autorizzazione" e non di "autenticazione". Infatti, può essere utilizzato per accedere alle risorse messe a disposizione senza necessariamente doversi identificare. Questo può causare molti problemi di sicurezza:

OAuth 2.0 non garantisce l'accesso ad informazioni necessarie per identificare una persona. Per quanto sia bello l'anonimato in rete, l'autenticazione è necessaria per associare un account ad un individuo, sia in modo che solo quell'individuo possa accedere alle sue informazioni e ai suoi dati di servizio, sia per avere una persona "legale" in caso di controversie nell'uso del servizio. L'indirizzo e-mail ad esempio è un mezzo comodo, in quanto serve anche ad avere un canale di comunicazione ufficiale con quell'individuo. Anche un identificativo, come un nome utente o un ID permetterebbe di riconoscere la persona, anche se in maniera parziale.

Implementazioni differenti: per risolvere il problema dell'autenticazione, ogni servizio che usa OAuth 2.0 ha implementato un proprio sistema. Questo è problematico perché crea disomogeneità. Le dirette conseguenze sono la scarsa interoperabilità dei sistemi di autenticazione, l'occorrenza di errori di implementazione, se si decidesse di copiare o imitare questi sistemi, e l'assenza di una linea comune su come quest'identificazione funzioni, anche per aspetti di sicurezza informatica o legali.

La soluzione è stata porre un ulteriore strumento a supporto di OAuth 2.0: OpenID Connect, un'estensione a OAuth 2.0 che colmasse quel piccolo gap che impedirebbe a OAuth 2.0 di risolvere il problema dell'autenticazione (29). Nonostante non sia utilizzato da entità come Google o Facebook, che ancora usano la loro implementazione di autenticazione su OAuth 2.0, permette di standardizzare l'autenticazione su OAuth 2.0. OpenID Connect prevede che, oltre all' "access token", venga fornito anche un ID Token. Questo ID Token permette di riconoscere come utente effettivo l'utilizzatore finale del servizio, di fatto permettendo di avere un "account" su quel servizio senza creare un account ad hoc, permettendo il concetto di Single Sign-On (SSO). Un solo account per multipli servizi. Solitamente, un ID Token può essere implementato tramite l'utilizzo di JWT. Inoltre, questo sistema accentra sull' "authorizing server", l'entità che concede l'autorizzazione al servizio per conto del client, diventa "identity server" a sua volta, permettendo di controllare gli ID Token.

Una nota importante di OpenID Connect è la standardizzazione di uno scope per prelevare informazioni utente: ciò significa che le informazioni richieste in fase di autenticazione, saranno uguali per tutti coloro che useranno OpenID Connect.

Nonostante sia una soluzione comoda e ben architettata, può soffrire di vulnerabilità anch'essa: essendo una soluzione basata su token, questi possono essere intercettati se non vengono fatte implementazioni corrette. La stessa specifica di OAuth segnala questi possibili problemi. Un ulteriore problema importante consiste nella sanitizzazione dei parametri del token di autenticazione e nel "redirect_uri", per evitare che codice malevolo venga iniettato attraverso questi parametri[4].

[4] https://tools.ietf.org/html/rfc6749#section-10.14, accenno al pericolo di Code Injection and Input validation, per cui si indica la sanitizzazione tutti i valori, con particolare attenzione a "redirect_uri" e "state"

2.8.2.4. Mutua autenticazione (mTLS)

TLS è una conoscenza più vicina, rispetto ai precedenti metodi, per creare canali che fossero sufficientemente sicuri per lo scambio di informazioni. Transport Layer Security (TLS) (30) è un protocollo che si appoggia su TCP/IP per garantire sicurezza in un canale di comunicazione, garantendo autenticità, riservatezza e integrità. E' così tanto utilizzato da essere alla base di HTTPS (HTTP over SSL/TLS). Nella sua semplicità, due tramite negoziano l'algoritmo da utilizzare per la cifratura della comunicazione, il protocollo di scambio chiavi e il metodo di autenticazione. In genere, all'inizio, il server si autentica presentando il certificato. Una volta validato, il client, in base al protocollo di scambio di chiavi (che può essere Challenge-Response o Diffie-Hellman o altri formati asimmetrici simili), avvia lo scambio di chiavi. Una volta scambiate le chiavi, vengono usate per cifrare la comunicazione.

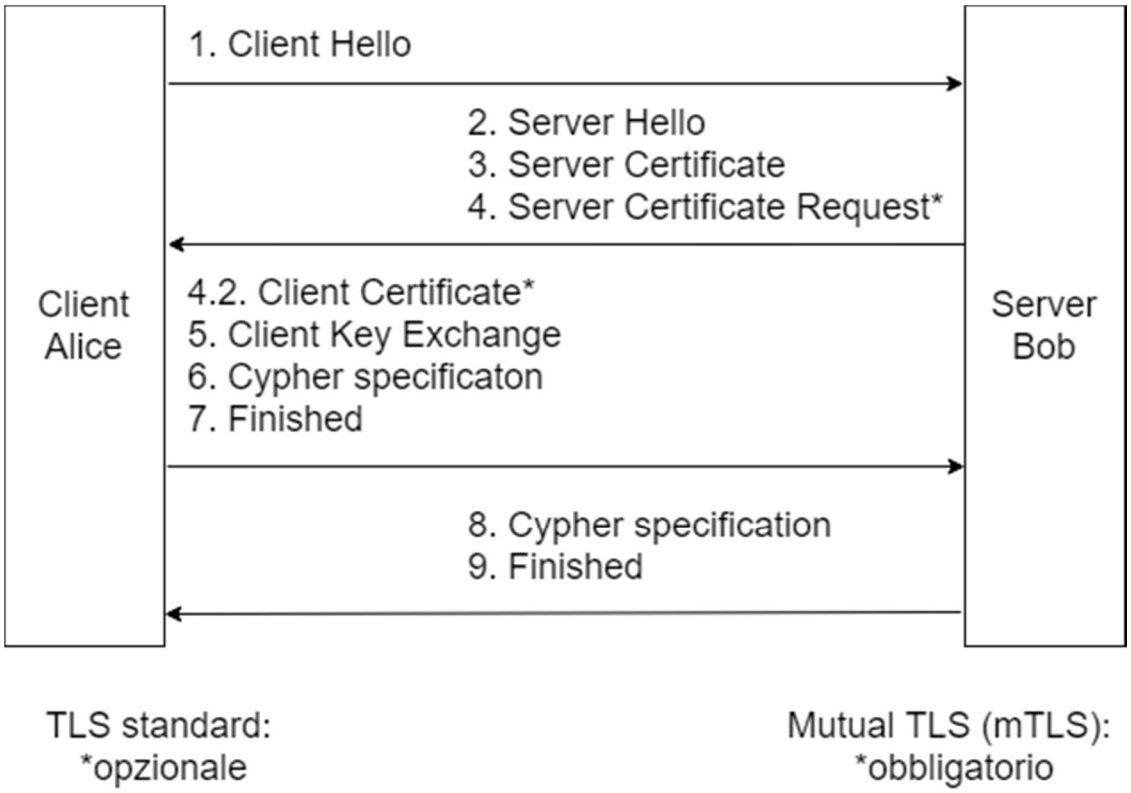

Figura 9 – La procedura di Handshake di TLS v 1.2, con il contenuto dei quattro messaggi "Client_Hello", "Server_Hello", "Client_Hello_Done", "Server_Hello_Done". Gli ultimi due comprendono, rispettivamente, i passaggi [4.2 -> 7] e [8, 9].

TLS, di base, prevede che il server si identifichi a priori, ma che l'identificazione del client sia opzionale. Questo è visibile anche dalla sequenza di messaggi scambiati nella procedura iniziale della comunicazione, guardando la Figura 9, cioè l'handshake: il client inizia con un messaggio "Client_Hello, che apre la comunicazione e indica al server le possibilità del client per la negoziazione dei parametri del protocollo. Il server può risponderà con "Server_Hello", in cui invia al client i parametri scelti. Successivamente,

invierà obbligatoriamente un messaggio “Certificate”, in cui manda il proprio certificato per incominciare lo scambio di chiavi, ma può inviare un “Certificate_request”, in cui si richiede al client di mandare il proprio certificato. Una volta inviato, il Client risponderà con lo scambio di chiavi verso il server e “Finished” segnalando di aver verificato il certificato. Se il server ha mandato il “Certificate_request”, manderà anche il suo messaggio “Certificate” per la verifica. Infine, il server invierà un messaggio “Server_Hello_Done”, con cui userà tutti i protocolli negoziati a conferma dell’avvenuto scambio di chiavi ed inizia la comunicazione.

Nonostante Mutual TLS, cioè TLS con l’autenticazione del client in aggiunta, renda TLS ancora più sicuro, da un punto di vista user-service, questo introduce una complessità notevole, cioè l’acquisizione da parte dell’utente di un Certificato, fornito e validato da una Certificate Authority (CA) ufficiale in genere esterna. La presenza di un ente esterno rende molto meno flessibile mTLS in termini d’uso (Monica 2016 – vedi Suo Malainen).

Considerando le difficoltà d’utilizzo per gli utenti, nell’ambito distribuito dei Microservizi può essere molto utile per quanto riguarda l’autenticazione tra servizi o tra macchine. Infatti, molti dei maggiori software di orchestrazione per Microservizi supportano l’uso di mTLS per l’autenticazione service-to-service. Ad esempio, Docker Swarmkit permette di stabilire una comunicazione mTLS usando CA interne e certificati x.509. Ciò diventa incredibilmente comodo, perché permette un’autenticazione relativamente sicura all’interno dell’organizzazione, con un contrappeso significativo: non avere una CA esterna significa diventare completamente responsabile sia della credibilità delle singole o della singola CA (inteso come “trust”), sia dell’integrità e sicurezza di tutta la Public Key Infrastructure (PKI) alla base del sistema. Ciò significa che se una CA viene violata, tutti i certificati validati da quella CA diventano non più affidabili.

Un altro problema maggiore di mTLS è che di per sé mTLS non ha come obiettivo l’autorizzazione: il protocollo è puramente d’autenticazione, perché si occupa di garantire standard di sicurezza sulla comunicazione. Infatti, solo nella versione TLS 1.2 furono inserite delle estensioni per l’autenticazione, esprimibili tramite regole SAML (31).

3. Sicurezza nei contesti Microservizi - Un approccio Blue Team

Sul tema dei Microservizi, ci sono molti studi e molte ricerche in merito. Tuttavia, non esiste ancora una teoria unificata o standard o modelli che descrivano le modalità di sviluppo di un sistema a Microservizi, tantomeno una teoria sulla sicurezza nei Microservizi. Un contributo maggiore a questa tesi, incentrata su una risposta quasi ad-hoc, viene da Tetiana Yarygina, che, attraverso diversi paper, ha provato a cercare indicazioni su come immaginare un modello di sicurezza per i Microservizi, per ottenere delle direzioni verso cui orientare il lavoro di ricerca. Come sostenuto da Yarygina, tuttavia, non esiste ancora un "modello", per cui molte considerazioni vengono fatte sulla base della struttura del progetto che si vuole costruire. Questa tesi, come si può evincere dagli obiettivi individuati nell'introduzione, non cerca di costruire un modello di sicurezza, in quanto sarebbe uno sforzo eccessivamente ampio, considerando tutte le tecnologie in gioco. Invece, l'argomento di questa tesi è sfruttare le conoscenze ottenute tramite le ricerche sull'argomento per individuare una strategia difensiva o un modello difensivo, nell'ambito dei Microservizi. L'obiettivo finale, come detto, è verificare la presenza o meno di un modello d'azione Blue Team da applicare, nel momento in cui il nostro sistema a Microservizi è sotto attacco.

3.1. Blue Team e Red Team – Il gioco della sicurezza

Il campo della sicurezza informatica è molto vasto e variegato, con tante sfumature e tante problematiche al suo interno, ma il problema di fondo di quella che è chiamata "Cyber Warfare" è costante: dato un sistema informatico, contenente delle risorse di interesse, può essere considerato un bersaglio da parte di individui od organizzazioni, il cui obiettivo è centrato attorno alle risorse d'interesse. La missione di queste parti rappresenta quella di impadronirsi delle risorse o del sistema, di disturbarne il funzionamento o di distruggerlo. Chi controlla il sistema informatico ha come interesse quello di difenderlo da queste parti. In termini di sicurezza, le parti il cui obiettivo è quello di ottenere un controllo sulle risorse rappresentano gli "attaccanti", mentre chi controlla il sistema e mira a proteggerlo è il "difensore".

Data questa divisione, una volta compresa l'entità del pericolo rappresentato da questi attaccanti, i compiti della sicurezza informatica sono stati quelli di trovare le falle, che permettono agli attaccanti di penetrare nel sistema informatico, di trovare le soluzioni che permettono di chiudere tali falle, di monitorare l'ambiente del sistema da proteggere e tenersi sempre informati su ogni sviluppo dello stesso. Questi compiti possono essere compiuti, anche contemporaneamente, dal "difensore", ma mancherebbe l'approccio "reale" di una situazione di attacco in corso e mancherebbe una "diversa ottica" che potrebbe trovare falle e soluzioni che, per il "difensore", potrebbero non essere visibili.

Da questo concetto e dalla pratica dell'esercizio militare, spesso chi si occupa di sicurezza informatica ha il compito di "simulare" attacchi informatici e ricavare quante più informazioni possibili, in modo da:

- Trovare falle da segnalare al team di sviluppo o da rattoppare.
- Trovare paradigmi offensivi con cui gli attaccanti possono operare: questi paradigmi rendono gli attacchi prevedibili, quindi facilmente individuabili quando un attaccante lancia un attacco al sistema informatico obiettivo.
- Progettare meccanismi automatici di prevenzione, individuazione, isolamento e risposta di possibili attacchi/attaccanti.

Detto ciò, si dividono gli attaccanti in quello che è definito il "Red Team" e i difensori nel "Blue Team".

In questa tesi, dunque, l'obiettivo è "impersonare" il Blue Team e capire tutte le possibili azioni che un difensore può compiere, nel contesto dei Microservizi, per reagire ad una situazione d'attacco e proteggere le risorse importanti.

3.2. Defense-in-depth – Difesa interna del sistema

Un altro concetto importante per comprendere questa tesi è il concetto di Defence-in-depth, o difesa in profondità.

Inizialmente, complice l'architettura monolitica, la difesa si concentrava sul perimetro esterno e il perimetro interno, ossia l'hardware della macchina, il luogo e l'organizzazione fisica erano considerati come "sicuri" o gestiti dall'organizzazione stessa. Successivamente però, con l'avanzare delle conoscenze informatiche degli attaccanti, la crescita esponenziale del modello distribuito (inteso come architettura), si sono verificati parecchi incidenti informatici che hanno mostrato come una sola difesa perimetrale non è sicura: non solo le componenti applicative possono avere delle vulnerabilità che permettono di scavalcarle, ma queste difese esterne sono minate anche da attacchi fisici, come tattiche di Social Engineering per ottenere credenziali all'interno dell'organizzazione bersaglio. Inoltre, sorpassato il sistema di difese esterno, si guadagnava una forma di controllo indiscriminata, senza misure di sicurezza che potessero quantomeno tamponare gli effetti di un'intrusione (32).

L'NSA, in risposta alla minaccia rappresentata dall'affidarsi ad un'unica difesa esterna, ideò il concetto di difesa in profondità (33) per fare in modo che non solo ci fosse un perimetro esterno, ma anche i singoli componenti esterni fossero protetti, in caso di compromissione o fallimento delle difese perimetrali. Il concetto di base è la stratificazione di misure difensive. Questi strati vengono applicati intorno ai componenti di un sistema, opportunamente individuati e definiti, per fare in modo che tra ogni componente vi sia una linea difensiva. L'obiettivo finale ovviamente non è prevenire un attacco a priori, ma ostacolare l'avanzata di un attaccante verso risorse critiche e limitare quanto più possibile i danni che un possibile intruso può compiere, una volta saltata la difesa perimetrale.

Alla base della defence-in-depth c'è il concetto di stratificazione delle difese: una volta individuate le risorse di un sistema informatico, si pongono strati crescenti di misure di sicurezza, partendo dall'hardware fisico e arrivando al perimetro esterno del sistema.

L'NSA individua tre entità che possono rappresentare vulnerabilità nella sicurezza:

- **"People" – Personale**: più che il personale umano, questa categoria fa riferimento al mondo fisico in generale. Solitamente, gli obiettivi in quest'area sono la formazione del personale, prevenzione nell'utilizzo sbagliato del sistema informatico, protezione dei dispositivi fisici, regolamenti e procedure adeguate, preparazione adeguata del personale addetto all'Amministrazione e Gestione del Sistema
- **"Technology" – Tecnologie**: la maggior parte degli attacchi informatici, vengono portati attraverso vulnerabilità rinvenute nelle tecnologie che si utilizzano e in come queste tecnologie cooperano e comunicano tra di loro. Oltre ad assicurarsi validazione ed affidabilità sulla sicurezza delle tecnologie in utilizzo, buona parte degli obiettivi sono quelli di organizzare sistemi stratificati, con capacità di protezione e di rilevamento delle intrusioni. In questo caso, si riferisce a IDS/IPS, firewall annidati, gestione robusta delle chiavi e della PKI, raccolta di informazioni per operazioni di rilevamento di intrusioni o direttamente di protezione.
- **"Operations" – Operazioni**: è la classe che racchiude tutte le attività effettive di sicurezza. Gestione di chiavi, certificati, utenti, stima e ricerca delle possibili vulnerabilità, garanzia e rispetto dei regolamenti di sicurezza (Security Policies), risposta ad attacchi e ripristino del sistema o delle sue componenti.

Per ogni categoria di vulnerabilità sono indicate linee guida, mezzi e sistemi di sicurezza. Inoltre, viene sottolineato come non siano aree distinte e sconnesse l'una dall'altra, ma che cooperano e che si snodano lungo tutto lo stack tecnologico di un sistema informatico.

Defence-in-depth diventa un modello necessario nell'ambito dei microservizi, in quanto il sistema interno non è centralizzato, i servizi comunicano tramite una rete internet e non un'intranet e i servizi possono essere quindi bersaglio di "altri servizi" che ovviamente sono stati compromessi. Con una superficie d'attacco che si allarga, è necessario prevedere misure di sicurezza multiple e stratificate, per fare in modo che una compromissione di un servizio o di un nodo non comporti la compromissione di tutto il sistema a microservizi.

3.3. Modello base di sistema a Microservizi – MicroBank

Per trovare una strategia difensiva, occorre analizzare un sistema a Microservizi, capire la struttura, le interazioni, ricercare i possibili attacchi e imbastire le difese in maniera adeguata. Per analizzare quindi un sistema a Microservizi, faremo riferimento al sistema "MicroBank"[5], sviluppato da Otterstad e Yarygina come "manichino" a Microservizi per PoC (Proof of Concept) atte a dimostrare la resistenza di un sistema a Microservizi ad exploit di basso livello (34).

Il sistema è strutturato come segue: sono presenti quattro componenti logiche.

- **Gateway**: componente che fornisce l'interfaccia utente, dando accesso alle funzionalità dell'applicazione.
- **Users**: servizio che si occupa di recuperare le informazioni sui singoli utenti e di effettuare operazioni di gestione.
- **Accounts**: servizio che si occupa di gestire gli account. Fornisce quindi gli account utenti e le operazioni di gestione.
- **Transactions**: servizio che si occupa di creare e gestire le transazioni effettuate dagli account.

Users, Accounts e Transactions, in quanto servizi, hanno il proprio database in cui vengono stoccati i dati, rispettivi per competenza (Users stoccherà nel proprio database le informazioni utenti, Accounts i dati dei singoli account bancari e Transaction stoccherà tutte le transazioni effettuate dagli utenti). Ogni servizio presenterà quindi un'interfaccia REST-like che permette di interagire con lo stesso. Quando, ad esempio, un utente richiederà una transazione verso un altro utente, Accounts andrà a comunicare con Users per individuare l'utente destinatario del movimento di denaro, effettuerà il movimento e comunicherà a Transaction il movimento effettuato per registrarlo.

In termini di tecnologie, MicroBank è scritta interamente in C, usa socket di tipo raw per pura semplicità e i database sono accessibili tramite SQLite.

[5] https://github.com/yarygina/MicroBank, MicroBank, T. Yarygina

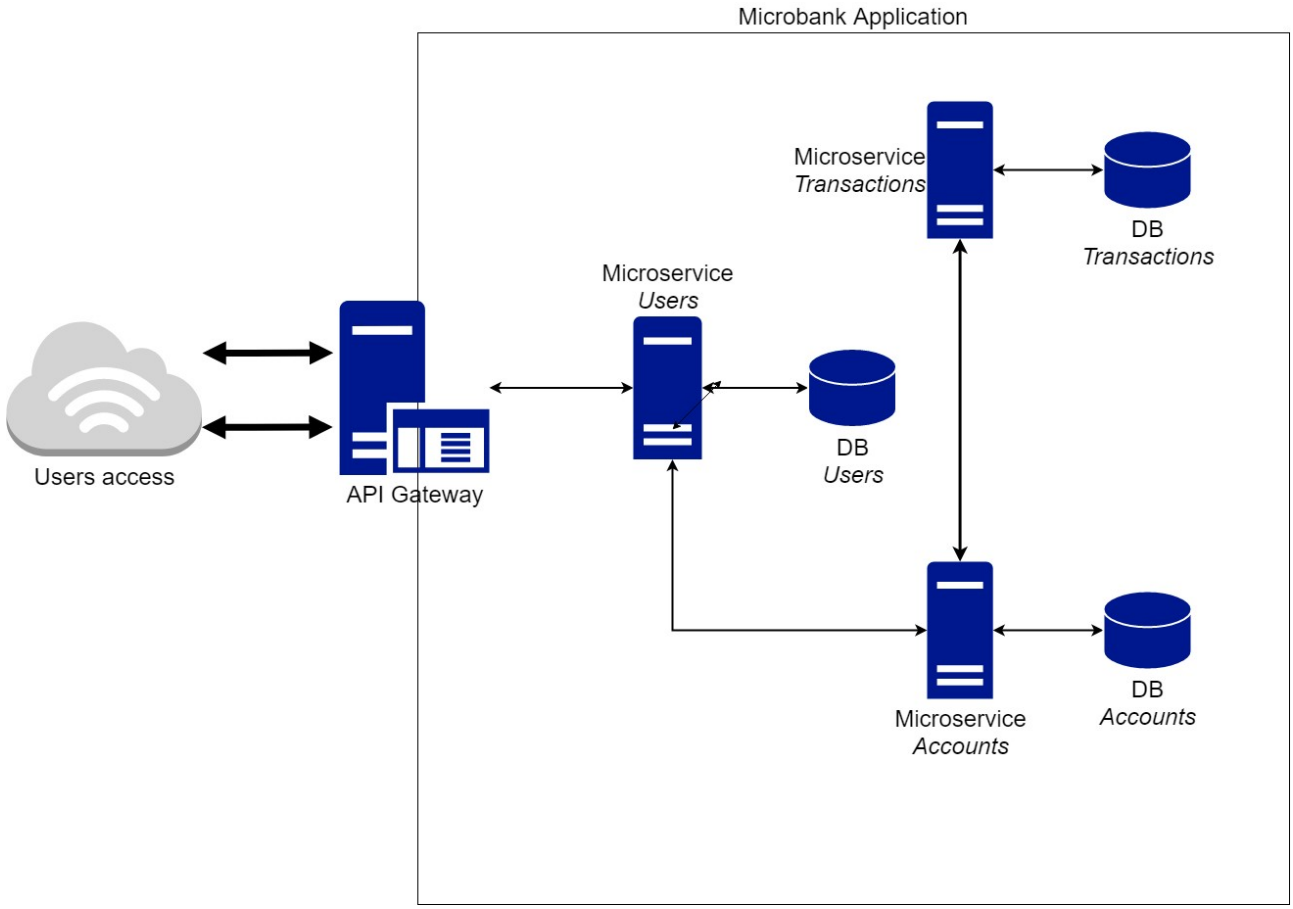

Figura 10 – Pseudo architettura dell'applicazione "MicroBank", di T. Yarygina.

In figura 10, è lo scheletro del sistema a Microservizi. E' importante osservare e analizzare lo scheletro, in quanto fin da subito emergono dettagli importanti per la sicurezza informatica del sistema.

Osserviamo, ad esempio, il flusso comunicativo, rappresentato in Figura 10: Accounts comunica con Users e con Transaction, Users comunica solo con Accounts, Transaction comunica solo con Accounts. Questo è importante, perché permette di identificare quali movimenti laterali può fare un attaccante. Osservare il flusso comunicativo tra servizi, permette di sapere attraverso quali servizi o componenti può passare l'attacco, quindi capire come si espanderà e dove sarà possibile, in caso, intercettarlo. Una volta intercettato, si può procedere all'isolamento, si risale alla catena dei servizi facenti parte del percorso dell'attaccante, respingere l'attaccante dall'interno e infine espellerlo dal sistema.

Inoltre, una caratteristica di base, sempre evidente dalla Figura 10 e dall'architettura dei sistemi a Microservizi, è che ci sono "due perimetri di sicurezza", non più uno solo:

- **Perimetro esterno**: rappresenta il "muro" che previene l'accesso indiscriminato all'interno del sistema. L'unico punto d'accesso esterno, sempre stando alla Figura 10, è quello rappresentato dall'API Gateway. Il Gateway non rappresenta solo l'interfaccia utente, che permette di usufruire dei servizi all'interno del sistema: implicitamente, è anche il "collo di bottiglia" attraverso cui passa tutto il traffico, che viene successivamente instradato all'interno, verso i servizi di riferimento. Per comprendere ancora meglio, il perimetro esterno di una intranet è in genere individuato dai gateway d'accesso all'intranet stessa e dal bastion host, che si pone come misura di sicurezza tra l'esterno e l'interno dell'intranet.
- **Perimetro interno**: poiché una delle caratteristiche dei sistemi a Microservizi è che i servizi possono trovarsi su host fisici diversi, questi comunicheranno attraverso la rete, secondo dei canali predisposti e fissati. Ma questi canali non sono interni, sfruttano ugualmente internet e come tali non possono essere monitorati direttamente. Ciò significa che ogni servizio deve tutelarsi anche dagli altri servizi e, in maniera grossolana, "vede" il traffico generato dagli altri servizi verso di sé come traffico esterno.

In Figura 10, il perimetro esterno è rappresentato dalla linea continua e spessa, mentre il perimetro interno è immediatamente visibile come linea tratteggiata attorno ai singoli servizi. Questa suddivisione di perimetri e le osservazioni sui flussi comunicativi fanno parte del concetto di "defence-in-depth". Il fatto che i servizi comunichino tramite internet e che abbiano quindi dei punti d'accesso, rappresenta una possibile falla sfruttabile in remoto. Questo permetterebbe ad un attaccante di muoversi "lateralmente", cioè spostare il suo attacco su altre macchine facenti parte del sistema. Un'altra importante osservazione è la questione della "fiducia". In teoria, facendo parte tutti dello stesso sistema, sarebbe scontato pensare che un servizio sia "fidato" per gli altri servizi: mettendoci nell'ottica di un servizio, ci aspettiamo solo traffico da macchine che ospitano altri servizi, quindi dovremmo poterci fidare del traffico in ingresso, delle operazioni che ci vengono richieste, dei dati che ci vengono passati e delle risorse che daremo, senza necessità di controllare che tali richieste siano legittime o che chi le faccia sia autorizzato. L'unica eccezione sarebbe solamente data dal traffico di informazioni provenienti dal Gateway, ossia dall'esterno.

Proprio su questo punto invece è necessario fare più attenzione del dovuto, perché le condizioni teorizzate dall'esempio precedente sono vere, tranne una: non possiamo sapere, ad ogni momento, se la richiesta per quel servizio e per quella specifica risorsa od operazione, sia legittimata in termini di "autorizzazioni".

Sappiamo solo che, chiunque faccia queste richieste o queste operazioni, le sta facendo tramite il servizio che agisce, in un certo senso, come "proxy"[6]. Dopo aver visto il modo con cui i componenti comunicano, un'altra delle problematiche maggiori all'interno di MicroBank, in termini di sicurezza, è la presenza di tre database: di per sé, la letteratura sulla sicurezza dei database e sugli exploit SQL-related è già più che ampia e non verrà discussa. Ciò che è importante, dal punto di vista di un difensore, è che ci sono tre database su tre macchine diverse, acceduti da tre servizi diversi. Questo significa che dovremmo tenere sotto controllo lo stato di tutti e tre i database, come gli accessi che vengono effettuati.

Se pensassimo ad un sistema reale, la necessità di controllarli e proteggerli tutti, deriva dal fatto che tutti e tre sono critici:

- Il db "Accounts" mantiene gli account utenti e le informazioni ivi contenute. E' scontato che sia di vitale importanza evitare accessi indiscriminati a questo database, a meno di non voler rendere pubblico il contenuto dei conti correnti dei propri clienti e l'identificativo degli utenti a cui questi conti correnti fanno conto.
- Il db "Users" mantiene le informazioni sugli utenti, tra cui anche informazioni personali, contatti e password.
- Il db "Transactions" non è critico per le informazioni direttamente accessibili da un'attaccante, visto che conterrebbe i resoconti delle transazioni effettuate da tutti gli utenti. E' di vitale importanza che non venga mai modificato o eliminato, poiché, ad esempio, si perderebbero informazioni di un possibile attacco che ha comportato uno spostamento di denaro da un conto ad un altro. In sostanza, è un db critico a livello informativo per l'amministrazione in generale: se per l'amministrazione di sistema permette di verificare la presenza di movimenti sospetti e avere informazioni su chi l'ha generati, l'amministrazione della banca ne ha bisogno a livello legale, nel caso in cui sia necessaria la verifica di transazioni legate ad attività illecite, e a livello gestionale, in quanto tramite i flussi di ingresso e uscita si assicura la consistenza delle informazioni sul conto corrente di un possibile utente.

In questo caso va considerato anche il come è stato costruito il sistema a livello di linguaggi e di sistemi operativi: MicroBank, ad esempio, è scritta totalmente in C e ciò comporta che se un exploit funziona su un nodo, questo funzionerà anche sugli altri. Stessa cosa si può dire se il sistema operativo che ospita i container è lo stesso per tutti i nodi. In questo caso, la difesa è più "semplice", ma il fallimento di un nodo comporta una possibile compromissione di tutti gli altri nodi, nonostante l'attaccante debba prima riuscire a muoversi lateralmente nella struttura, un'azione non triviale.

[6] Questo è un accenno al problema del "Confused deputy", successivamente approfondito al capitolo 3.7, pag. 50

Supponiamo invece che, in MicroBank, ogni servizio sia scritto con linguaggi diversi, abbia DBMS diversi e sistemi operativi diversi (diversità tecnologica, uno dei punti cardine dell'approccio a microservizi): a livello di amministrazione sarebbe molto complesso riuscire a tenere sotto controllo e al sicuro ogni singolo database, ma questa complessità intrinseca vale anche per l'attaccante. Significa che l'attacco diventerà più complesso, dovendo utilizzare tecniche diverse e cambiando exploit ad ogni nodo. Un'attaccante più lento dà una finestra temporale più ampia per il difensore, che può intervenire tempestivamente ed isolare l'attaccante sul primo nodo infetto.

3.4. Tanti servizi, maggiore superficie d'attacco

Come si evince dalla necessità dietro il modello defence-in-depth, dalle considerazioni iniziali su MicroBank e aggiungendo anche le caratteristiche dei modelli a microservizi, si può concludere che, rispetto al modello monolitico, la superficie d'attacco è decisamente variabile. Per capirlo, basta confrontare la tassonomia individuata da Yarygina (3) [7] delle possibili vulnerabilità e i risultati della PoC di Yarygina[8] e Otterstad (34). Inoltre, nonostante si presume che i tre microservizi di Microbank fossero in locale, sulla stessa macchina, in genere i microservizi trovano la loro maggiore efficienza quando sono usati in combinazione con la tecnologia "cloud". Più in generale, quindi, i microservizi non saranno fisicamente sullo stesso host, ma saranno su macchine diverse. E' ragionevole pensare anche che siano a distanza remota, su cloud provider diversi o su sedi fisiche diverse.

Riprendendo la Figura 10 del modello di MicroBank: sebbene rappresenti una possibile configurazione, è già possibile intuire come, rispetto ad un'applicazione tradizionale, l'approccio a Microservizi cambi drasticamente la superficie d'attacco possibile, sia positivamente che negativamente.

3.4.1. La tassonomia nei Microservizi

Prima di procedere con l'esame della superficie d'attacco, conviene trovare una schematizzazione di quelli che sono i livelli in cui una struttura a microservizi si divide. Oltre ai livelli delle architetture precedenti, che prendono ispirazione dal modello OSI, si aggiungono nuovi livelli operativi che rispecchiano nel nuove tecnologie. Riprendendo quindi i concetti in tal senso delineati da Yarygina, Bagge e successivamente da Suomalainen (35) [9], abbiamo i seguenti strati in cui è possibile decomporre un sistema a microservizi: Hardware, Virtualizzazione, Cloud, Comunicazione, Livello Applicativo e Orchestrazione. Come si può notare, Virtualizzazione, Cloud e Orchestrazione sono livelli aggiunti in funzione delle nuove tecnologie utilizzate attualmente, soprattutto nell'ambito dei microservizi. Grazie a questa tassonomia, è possibile individuare una superficie d'attacco generale ed iniziale, a cui poi vanno affiancati i casi particolari per avere una più chiara rappresentazione della reale superficie, per analisi efficienti.

[7] Cap. 4.1, Tabella 4.1, p.60
[8] Da qui in poi, per semplificare, l'espressione "PoC di Yarygina" riferirà al paper citato come (34)
[9] Cap. 2.3, Tabella 1, p. 9

Layers	*Descrizione*
Hardware	Alla base di ogni astrazione, rimane un problema a causa di bug (vedasi Meltdown e Spectre) o di manomissioni.
Virtualizzazione	Macchine virtuali e Container offrono diversi approcci, ma stesse minacce. Sandbox escape, Hypervisor/Container Engine compromessi, attacchi sulla memoria condivisa e l'uso (o la presenza) di Immagini Virtuali o Container pericolosi possono rappresentare pericoli importanti.
Cloud	Diventata una base per i microservizi, il Cloud rappresenta un pericolo in quanto non è controllabile direttamente, dato che il controllo rimane nelle mani del provider di servizi Cloud.
Comunicazione	Non solo le comunicazioni con l'esterno, ma anche le comunicazioni tra servizi possono diventare problematiche in termini di sicurezza. Una buona notizia è che le meccaniche offensive non cambiano, in termini di comunicazione, e si può fare riferimento ai modelli di sicurezza classici (Sniffing, Spoofing, DDoS, MITM e attacchi ai protocolli, come POODLE e Heartbleed per TLS).
Livello applicativo	Il livello applicativo non cambia, rimangono problemi tipici che possono variare dall'implementazione errata di determinate meccaniche (SQL Injection, XSS, autenticazione e autorizzazione configurate male) ad errori di configurazione dei moduli di sicurezza. Come non cambiano i problemi, non cambiano i modelli di sicurezza e le problematiche (le indicazioni di minacce e rischi dell'OWASP ne sono un chiaro esempio)
Orchestrazione	L'amministrazione dei servizi, come la coordinazione e l'automatizzazione di operazioni, può essere problematica, considerando come la struttura di una rete a microservizi possa cambiare di continuo, con servizi che vengono avviati, stoppati e sostituiti. Proteggere quindi i componenti d'orchestrazione e i meccanismi con cui coordinano i servizi diventa essenziale, per evitare una compromissione su larga scala.

Tabella 1 - Tassonomia nei microservizi, rielaborata a partire dalla tassonomia di T. Yarygina e J. Suomalainen.

Questa tassonomia risulta molto importante per capire quale sia la superficie d'attacco, se applicata ad un sistema. In un'ottica Blue Team, che esclude quindi la realizzazione e lo sviluppo dell'architettura, la tassonomia si riduce, permettendo di focalizzare meglio gli ambiti in cui muoversi per analizzare le potenziali minacce e delineare uno schema del sistema più preciso su cui muoversi.

Non solo, la tassonomia si riduce inevitabilmente anche per l'impossibilità, in alcuni casi, di intervento diretto da parte del Blue Team. Ad esempio, il layer hardware è parzialmente coperto: nonostante si possano mettere in uso Hardware Security Modules, il Blue Team non può rispondere completamente ad una minaccia hardware. Sempre Meltdown e Spectre hanno dimostrato quali sono i limiti per un team di sicurezza, che può fare poco se non raccogliere informazioni (36) sulle minacce, come riconoscerle e attendere che le case produttrici rilascino aggiornamenti o nuove componenti hardware resistenti a questo tipo di attacchi.

Anche il livello applicativo limita il lavoro di un Blue Team: poiché spesso non sono coinvolti direttamente con il processo di sviluppo, specie in un ambito come i Microservizi (stesso discorso vale per i sistemi distribuiti e per l'SOA in generale) e in modelli di sviluppo come le DevOps, non possono intervenire direttamente. Questo ha sollevato la necessità di modificare l'approccio DevOps, enfatizzando, ad esempio, la cooperazione tra DevOps e personale di sicurezza (37), per cui il concetto si sta evolvendo verso quello che è chiamato "DevSecOps" (38). Per quanto riguarda il Blue Team, è fondamentale quindi che ci sia un continuo flusso di informazioni sul processo di sviluppo, per individuare possibili falle per cui tenere gli occhi aperti e su cui orientare meglio gli strumenti di monitoraggio.

3.4.2. Il perimetro di sicurezza

Una questione apparentemente triviale è definire il perimetro di sicurezza. Normalmente, seguendo lo schema classico delle applicazioni Web, il perimetro di sicurezza risulta essere un "rettangolo" che separa il server e l'applicazione dal mondo esterno. "Esterno" è un termine significativo: il perimetro di sicurezza identifica la minaccia all'esterno del sistema, non al suo interno. Questo modello è anche chiamato "Castle model" o "Fortress-based security" (39) è sicuro, perché separato dall'esterno dalle "mura" e dalle fortificazioni. Sui bordi del sistema possiamo avere un firewall, che effettua un primo passo di filtraggio del traffico in ingresso: regoliamo quindi "chi e cosa" può "accedere alla fortezza". Non solo, vengono posti sistemi di autenticazione sulle mura, per fare in modo di controllare chi ha il diritto di accedere e chi no.

Tuttavia, la semplice difesa perimetrale non basta. Se il nemico è capace di abbattere o scavalcare le mura, c'è bisogno di difendere anche dall'interno. La strategia si evolve per proteggere anche all'interno, mettendo in piedi sistemi e meccanismi di sicurezza. Per questo si ricorre agli IDS (Intrusion Detection Systems) e, successivamente, gli IPS (Intrusion Prevention Systems). I primi rappresentato meccanismi di controllo, che riconoscono attività sospette ed emettono "allarmi" in merito: abbiamo ad esempio Swatch e LogWatch, che permettono di analizzare i log per verificare se ci sono anomalie su cui intervenire e di proteggere i log stessi, Tripwire e AIDE, che si occupano di

monitorare l'integrità dei file, Snort, che si occupa di analizzare il traffico in cerca di operazioni sospette, come scanning.

I secondi invece sono più servizi, che semplici meccanismi: non solo si occupano di fare operazioni simili agli IDS, ma intervengono direttamente usando gli strumenti di controllo e amministrazione per effettuare operazioni di correzione o di opposizione a tentativi d'attacco. Esempi sono Splunk (traffic-related) e OSSEC (HIDS completo, con capacità reattive), sistemi più complessi che non forniscono solo funzionalità IDS, ma anche capacità di reazione, intervenendo direttamente sul problema. Ad esempio: in caso di traffico sospetto, si blocca l'indirizzo all'accesso dei servizi in risposta al possibile sospetto di operazioni "pericolose", come la scansione delle porte.

Questo modello presenta però falle, sia naturalmente che applicato a nuovi contesti tecnologici, come i sistemi distribuiti e, ancor più, i microservizi:

- **Il fattore umano o "personnel"**: come segnalato anche dalla teoria dietro il concetto di "Defence-in-depth", uno dei principali punti deboli dell'approccio "Fortress-based" sta nel fatto che il nemico può essere all'interno, intenzionalmente o meno. Un impiegato potrebbe manomettere dall'interno il sistema o cedere le proprie credenziali d'accesso, che le ceda volontariamente o venga ingannato a farlo. Tradotto: il nemico è all'interno delle mura e non è riconosciuto come tale.
- **Decentralizzazione e distribuzione**: non abbiamo più "un solo castello", ma abbiamo più castelli da tenere d'occhio. Non si tratta quindi di concentrare i propri sforzi su un solo sistema, ma su più sistemi contemporaneamente, che non evolvono parallelamente in maniera automatica. Lo sforzo di decentralizzazione dei servizi web ha portato anche ad avere più sistemi diversi da proteggere, sistemi che possono differire in termini di tecnologie impiegate, business logic, ragioni strategiche. In sostanza, hanno diverse priorità e diversi strumenti con cui essere protetti a disposizione, in genere non sempre coincidenti tra loro.
- **Standard di performance e interoperabilità** (40): abbiamo visto che i servizi web diventano sempre più interconnessi tra loro, come la possibilità di usare un servizio per autenticarsi presso altri servizi, e come siano diventati importanti standard di perfomance, soprattutto collegati alla disponibilità e alla rapidità di un sistema. Performance che con un "muro", sebbene necessario, non si riescono a raggiungere.
- **Fiducia**: con le precedenti informazioni sull'autenticazione e l'autorizzazione, è possibile delineare un problema di fiducia iniziale, nel mondo esterno al sistema. All'interno però, tutti i componenti sono considerati sicuri, come anche il riconoscimento dell'utente tramite autenticazione e autorizzazione renda "fidato" l'utente. Tuttavia, il dubbio sulla "fiducia" della prova d'identità rimane. Per le macchine il discorso cambia? Certamente, in quanto all'interno di un sistema una macchina potrebbe ospitare un attaccante e comportarsi normalmente, apparendo al sistema come "normale" e quindi fidata a priori.

Quindi, in un mondo di Microservizi, dove i servizi sono entità distaccate e distanti l'una dall'altra e dove un'applicazione non ha una "forma" compatta, qual è la nuova definizione di perimetro?

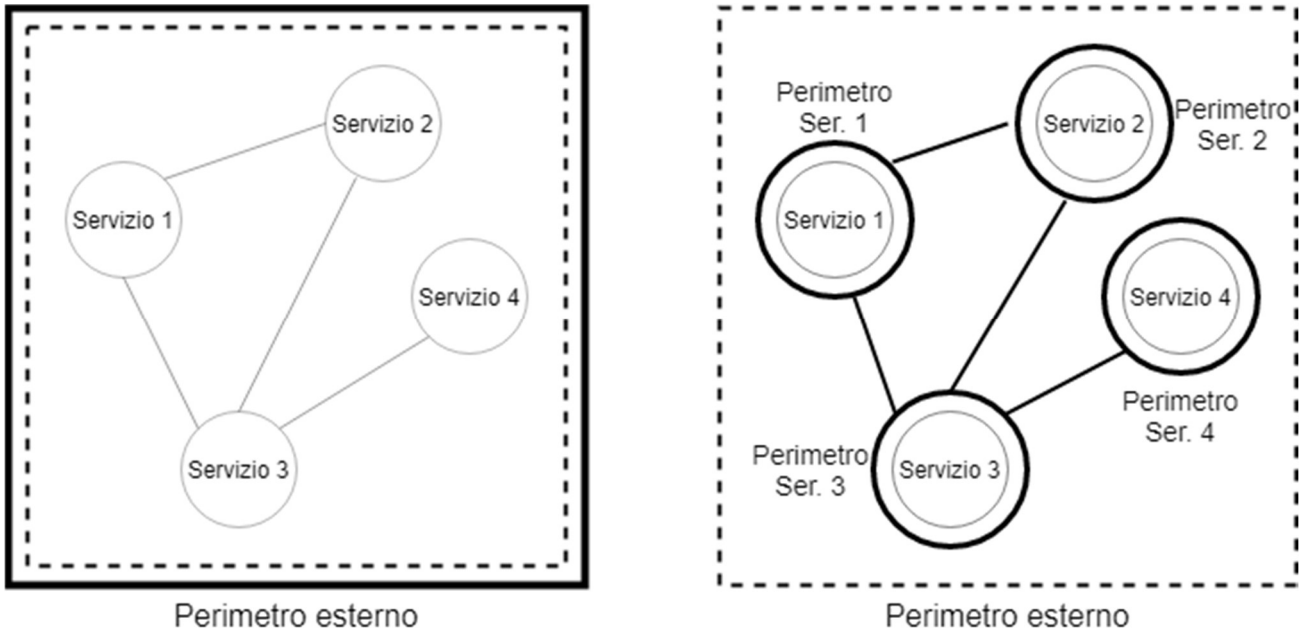

Figura 11 - Dal semplice perimetro esterno ad una struttura a più perimetri, dove c'è un perimetro per ogni componente e, in aggiunta, un perimetro esterno. Il perimetro esterno nuovo isola e screma tutto ciò che proviene dall'esterno, mentre ogni singolo servizio dispone di un perimetro difensivo specifico per quel servizio.

Non più un unico perimetro esterno, ma ogni servizio ha il proprio "perimetro". Com'è ben evidente dalla Figura 11, il singolo servizio, nell'applicazione, ha un proprio perimetro, che non è aperto nei canali di comunicazione verso gli altri servizi. Anzi, il perimetro "personale" dei servizi serve a proteggerli anche da altri servizi, evitando uno spostamento laterale dell'attaccante all'interno dell'applicazione ed evitando di rendere accessibile tutta l'applicazione all'attaccante.

Ancor di più se si pensa che nel mondo dei Microservizi le tecnologie di sviluppo sono eterogenee: ogni servizio può essere scritto in un linguaggio diverso, basato su tecnologie diverse e comunicante con protocolli o modalità differenti rispetto ad altri servizi. Tale varietà, rafforzata dal concetto di un perimetro "di servizio", permette di rendere veramente resistenti applicazioni basate sui microservizi. Per contro, tuttavia, il pericolo viene concentrato nelle API di comunicazione che il servizio espone all'esterno e sui meccanismi con cui i servizi comunicano tra di loro. ESB, entità pub-sub, servizi di discovery per altri servizi, rappresentano potenziali minacce "esterne", perché non direttamente collegate con i servizi e quindi su cui non si hanno informazioni per stabilire una "fiducia". In questo caso, il trade-off è comunque vantaggioso, diminuendo la superficie d'attacco a livello macroscopico.

3.5. La mappa di un sistema a microservizi: evidenziare la superficie e definire il modello.

Una volta definita una tassonomia dei microservizi e una ridefinizione del perimetro di sicurezza, risulta più agevole effettuare uno schema accurato di un possibile sistema a microservizi per un Blue Team. Per proseguire con un approccio più pragmatico agli esempi, delineiamo un sistema sfruttando il sistema d'esempio MicroBank. Oltre ai tre Microservizi e all'API Gateway:

Aggiungiamo un "Identity provider": un componente a supporto dell'API Gateway, che permette di utilizzare OAuth 2.0 + ID Connect (tramite token JWT), per autenticare e autorizzare gli utenti all'accesso e a supporto di Docker Swarm, contenendo anche la CA/KPI per fornire certificati ai container.

- I servizi saranno contenuti in container e gestiti tramite Docker Swarm.
- Un "service registry" che permette alle singole istanze di servizi di essere visibili e di fare discovery per il Gateway e per i singoli servizi.
- Ogni "host" avrà repliche dello stesso servizio contenute in containers.

Questo per renderlo quanto più vicino possibile ad una reale struttura microservizi. Considerando che il discorso verte sull'analisi del lavoro di un Blue Team, nello schema non sarà inclusa la catena di sviluppo, ma solo l'"applicazione" in esame.

Lo schema di MicroBank si trasformerà in una struttura simile alla figura successiva.

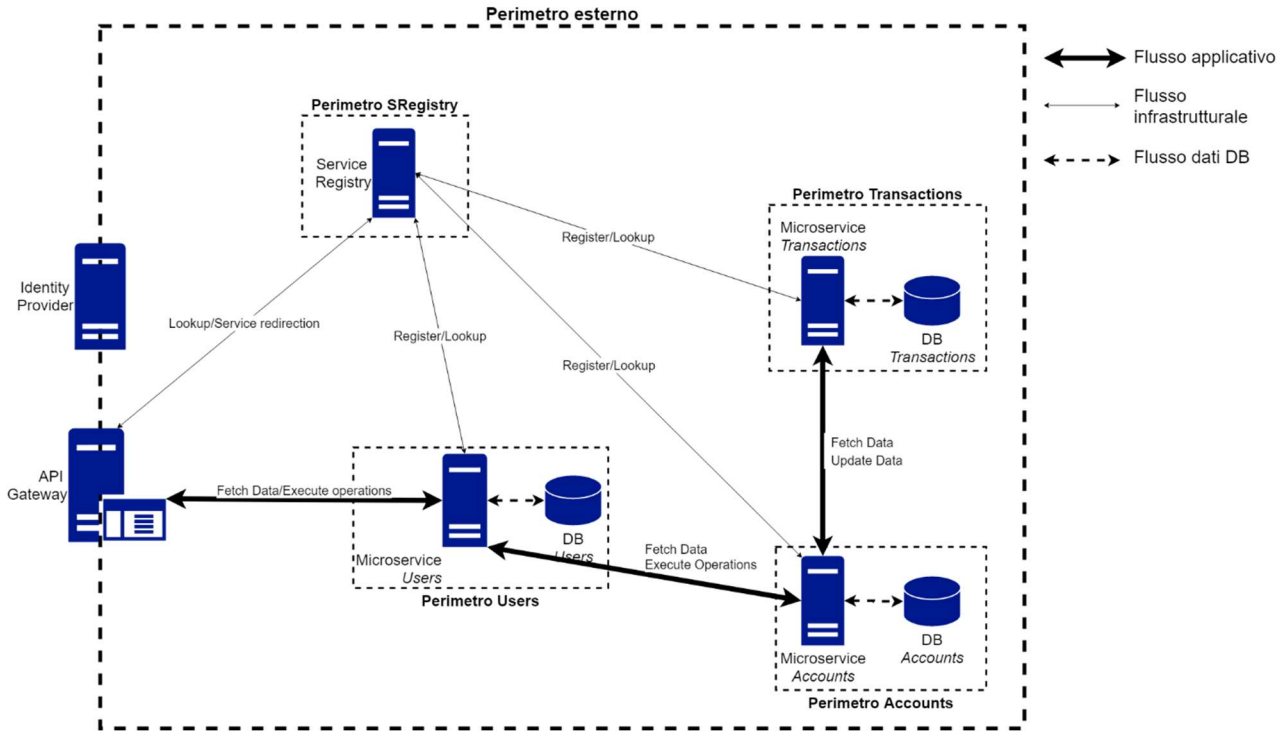

Figura 12 - Modello ideale del sistema MicroBank. In evidenza non solo il Service Registry e l'Identity Provider, ma anche i perimetri interni dei singoli servizi.

Oltre alla spiegazione precedentemente fornita circa MicroBank, possiamo notare quindi che:

- L'utente passerà dall'Identity provider per farsi autenticare e autorizzare, per poi presentare i token al Gateway
- Il service registry indirizzerà il gateway ad uno dei container del servizio richiesto.

Considerando che la difesa esterna, concernente API Gateway e Identity provider, sia un argomento già più che ampliamente discusso dalla bibliografia in termini di sicurezza informatica, le seguenti osservazioni riguarderanno il mondo interno dei Microservizi, in particolare la differenza introdotta dalla containerizzazione nella singola macchina e la comunicazione intra-server.

3.6. Il singolo server

Anche se la forma dell'app cambia e diventa una rete di singole "applicazioni" che formano un servizio, rimane costante la struttura di base del singolo servizio. Abbiamo quindi un server, che contiene l'applicazione di gestione dei container (Docker, in caso), il sistema operativo e l'infrastruttura. In questo senso, possiamo vedere immediatamente l'anatomia delle macchine a confronto.

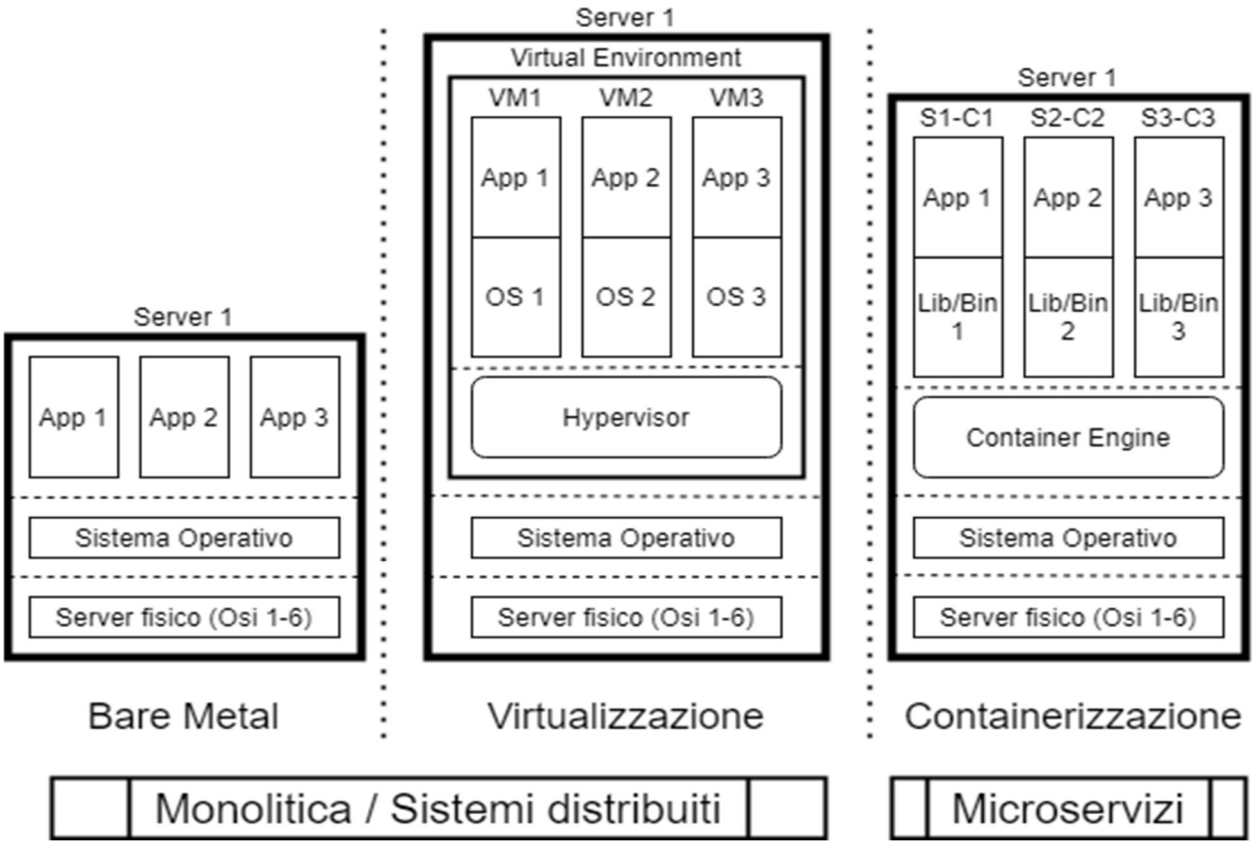

Figura 13 - Strutture dei server standard, con virtualizzazione, con containerizzazione.

La differenza maggiore comportata dalla containerizzazione è la seguente: guardando la figura, nella configurazione "Bare Metal" le app poggiano direttamente sul sistema operativo, per cui il codice può influenzare direttamente i livelli sottostanti.

Fin qui, siamo nell'ambito della base della sicurezza informatica. Nella configurazione "Virtualizzazione", invece, le app vengono chiuse in ambienti isolati, le Macchine Virtuali (VM): per cui, queste vengono separate dal sistema sottostante, il cui unico punto di contatto tra interno ed esterno diventa l'Hypervisor, che funziona da "traduttore" per i sistemi operativi delle macchine virtuali verso la macchina fisica. La letteratura in merito a quanto la virtualizzazione sia pericolosa è abbondante, al punto di fornire mappe molto dettagliate in merito (41).

Mentre nella Virtualizzazione abbiamo un ambiente presumibilmente distaccato rispetto al sistema operativo, nella Containerizzazione invece il sistema virtuale poggia sul sistema operativo stesso, isolando non delle intere macchine, ma semplici applicazioni. Questo risolve molti dei problemi creati dall'avere un secondo sistema operativo sotto l'applicazione (avere doppie vulnerabilità), che può essere sfruttato grazie al funzionamento dell'hypervisor in un attacco VM Escape. Il Container poi contiene solo l'applicazione, binari e librerie, diminuendo drasticamente le vulnerabilità.

Il punto in comune rimane l'anatomia sottostante. Per i threat models e per le minacce si può sempre fare riferimento, in questo caso, a modelli come STRIDE (cit. necessaria). Discorso differente va fatto per i Container e per il Container Engine, che però non si spostano più di tanto, rispetto alla virtualizzazione.

Superficie d'attacco	Vettore d'attacco	Minacce / attacchi
Intra-container	• Applicazione vulnerabile • Container malevolo	• Container escape • Container image attack • Root-kernel attack • Config tampering • Sensitive information leak
Extra-container	• Engine compromesso • SO compromesso • Insider attack/backdoor	• Container Sprawl • Service spoofing • Data leak • Backdoor

Tabella 2 - Possibile tassonomia di sicurezza per il singolo server in ambito Microservizi.

La lista di attacchi in questa tassonomia non è esaustiva. I vettori, così come gli attacchi indicati, indicano degli scenari di attacchi più grandi, di cui poche sono delle vulnerabilità generali registrate (CVD, per esempio). Molto spesso infatti, le vulnerabilità sono strettamente legate alla tecnologia in utilizzo, non tanto ai modelli in utilizzo. Ovviamente, questa tassonomia permette di indirizzarsi verso la ricerca di determinate vulnerabilità, con l'ausilio di OpenVAS o altri sistemi simili di scansione.

3.6.1. Esempio di Container Escape: cgroup e container privilegiati

Un esempio di container escape viene da una vulnerabilità recente scoperta in come i container e cgroup, l'utility Linux alla base di qualsiasi sistema a container, lavorano insieme. Questa vulnerabilità vale sia per i container Docker che per i pod Kubernetes, diventando di carattere generale. La vulnerabilità consiste nell'attivare container con flag di sicurezza elevati, come –privileged o –cap-add=SYS_ADMIN, ossia di dare capabilities o poteri privilegiati al container, e sfruttare la funzione di cgroup "notify_on_release" per l'esecuzione di codice arbitrario con privilegi root. Questo deriva dalla struttura dei container, che sono come dei processi, controllati da un "utente" nel sistema e inseriti in un ambiente simile, ma separato rispetto al sistema di base. Necessitando di un sistema di controllo e gestione delle risorse, cgroup risulta perfetto per il compito, senza avere necessità di modificarne il comportamento di base. Il movimento è schematizzato come segue:

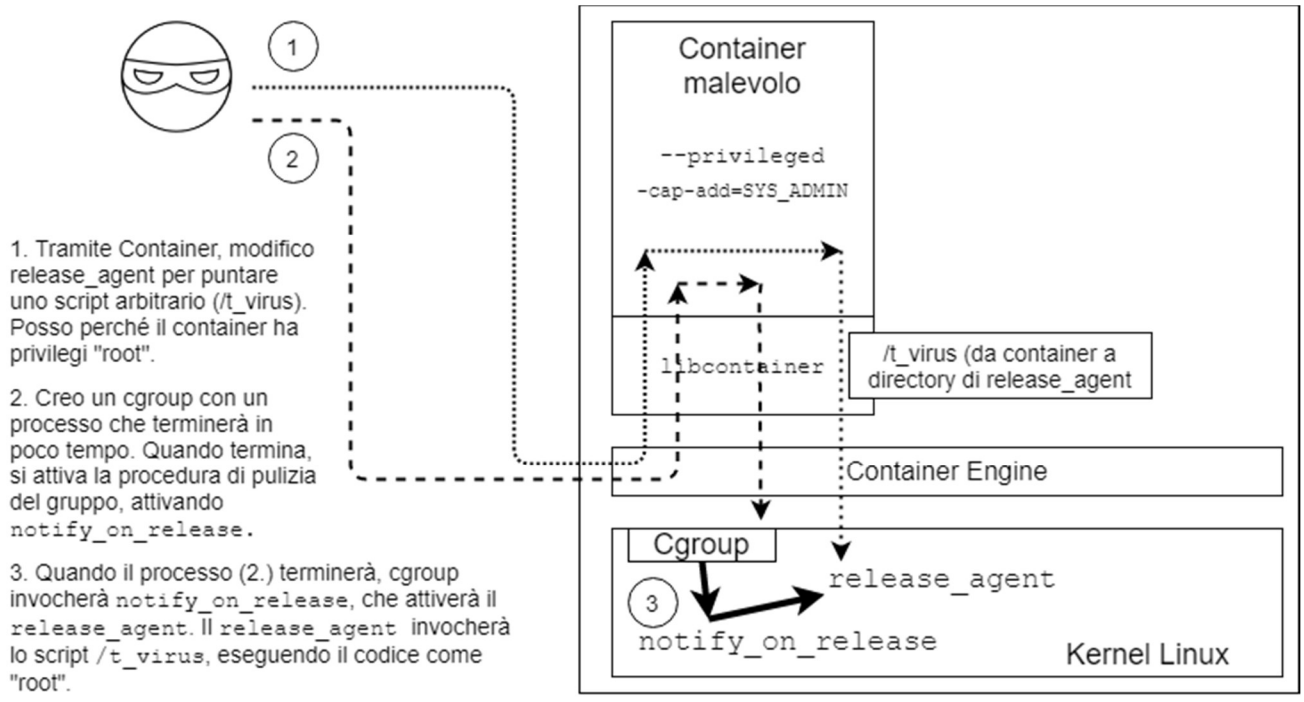

Figura 14 - Una schematizzazione dell'attacco descritto. E' interessante, nello schema, vedere come tutte le comunicazioni passino per il Container Engine, che dispone la visibilità del servizio di containerizzazione all'esterno. La fase 3 dell'attacco, avviene in maniera automatica, senza che l'attaccante debba interagire direttamente con il sistema.

Dalla Figura 14 si nota che un container privilegiato può sfruttare il collegamento tra il componente libcontainer e il kernel Linux per modificare "release_agent", un file che viene eseguito quando l'ultimo task in esecuzione lascia il cgroup. Normalmente, il "release_agent" serve a ripulire in automatico i cgroup abbandonati, per evitare che rimangano attivi e consumino o trattengano risorse. Però, "release_agent" è invocato con privilegi root: se ne viene cambiato il comportamento, nulla può fermarne l'esecuzione.

Quindi, un container privilegiato può eseguire operazioni di mounting, creare un cgroup apposito e modificare il codice di "release_agent" per richiamare un file contenente il codice malevolo. Tutto ciò può essere fatto direttamente sull'host, usando il container come "base operativa" e permettendo quindi di muoversi all'interno dell'host stesso. Fortunatamente, gestire in maniera appropriata i permessi e i privilegi con cui un container viene avviato, permette di evitare queste minacce, come indicato nell'articolo d'esame di questa vulnerabilità (42). Questa vulnerabilità può essere anche usata in combinazione con più vettori d'attacco, per orchestrare un attacco più complesso ed efficace (43).

3.6.2. Difendersi nell'ambiente containerizzato

Una nota positiva è che questi attacchi ai container sono più facilmente gestibili: i container sono entità così flessibili ed effimere, che l'eliminazione e il ripristino di un container è triviale, finché si hanno immagini sicure. Non solo, anche il downgrading non risulta un problema, in quanto l'aggiornamento di un servizio contenuto in un container non risulta essere un'operazione lunga e complessa come l'aggiornamento di un intero server. Infatti, è prassi che i container vengano spesso abbattuti, sostituiti e ricostruiti.

Nel paper di ricerca "A Game of Microservices: Automated Intrusion Response", di Otterstad & Yarygina (44), l'obiettivo è quello di analizzare e ragionare su un sistema automatizzato di difesa, delineando le mosse del difensore in relazione alle mosse dell'attaccante e alle disponibilità di sistema. Nell'ambito dei Microservizi, in risposta ad una intrusione, si hanno molte mosse disponibili, grazie alla grande flessibilità dei container, per gestire possibili compromissioni di servizi o nodi. Delle mosse indicate, le mosse principali da considerare possono essere:

- **Rollback / Riavvio del servizio**: si distrugge l'istanza del servizio e se ne genera una nuova, a partire dall'ultima configurazione ritenuta sicura (che può essere la precedente versione). Il fatto di poter agilmente distruggere un container per crearne un altro, a patto di avere immagini di container sicure, permette di rimuovere la presenza dell'attaccante. Se l'attaccante è penetrato direttamente nel nodo senza curarsi di creare una via d'accesso che non utilizzi il container d'ingresso, in questo modo l'attaccante è quantomeno obbligato a ricominciare da capo, esponendosi al riconoscimento ad un secondo tentativo.
- **Diversificazione:** ci sono due diversificazioni che si possono fare, se non si hanno a disposizione più versioni di uno stesso servizio, secondo il concetto di N-version Programming (45). In prima istanza, si possono usare speciali compilatori per diversificare i binari che eseguono nel container. Similmente, è possibile usare compilatori che riscrivono i binari per differenziarli. In ultima istanza, si può ricorrere allo spostare un microservizio su un host differente (anche un Cloud differente), per impedire all'attaccante di poter ripetere immediatamente l'attacco e di forzarlo a cercare nuovamente il container d'accesso.

C'è un caveat che viene fatto da Yaryjina nello stesso studio, ma su un paper differente (34) [10]: parlando di un teorico "Security monitor service", nell'ambito della mitigazione degli attacchi low-level: nel caso di utilizzo di queste tecniche per distruggere il servizio compromesso, è necessario effettuare particolare attenzione nel distruggere l'intero ambiente in cui eseguiva il servizio, per evitare che l'attaccante possa sopravvivere alla pulizia del servizio.

[10] Cap. 5.4.1, p. 87

Per comprendere meglio, i casi posti sono due:

- **In assenza di container/macchine virtuali**: non solo tutti i processi vanno eliminati, ma anche tutto il sistema al di sotto. Quindi si procede a ripristinare sistema e firmware a partire da un'immagine fidata del nodo, per poi riattivare i servizi.
- **In presenza di container/macchine virtuali**: l'intero container o macchina virtuale deve essere cancellata e ricostruita a partire da un'immagine container o VM pulita.

Come si può evincere dalla descrizione del caveat, l'uso dei container rende molto più semplice l'operazione, mantenendo comunque un possibile requisito di high availability, se ci sono più repliche del servizio stesso. In caso tutti i servizi siano compromessi, è molto più agevole, nel secondo caso, distruggere e ricreare container che macchine virtuali. Rispetto ai container, l'assenza di molti componenti rende le operazioni meno onerose e lunghe.

3.7. Comunicazione intra-servizi

Mentre un'applicazione monolitica ha tutto dentro lo stesso server, qui invece abbiamo più server che comunicano tra loro all'interno di un ideale "perimetro" per cui viene definita l'applicazione. La differenza è infatti che i servizi non comunicano con l'esterno, comunicano solo tra loro e con l'API Gateway, di base. Ci sono anche altre componenti, come il servizio di discovery, il servizio di CA e l'Identity Provider, con cui potrebbero comunicare, per ragioni infrastrutturali. In virtù del fatto che un nodo possa essere compromesso, è stato necessario rivedere il reale perimetro di sicurezza secondo il modello Zero Trust, in quanto è ragionevole assumere che qualsiasi comunicazione in ingresso possa essere ostile. Inoltre, la struttura a Microservizi porta con sé problematiche nuove, alcune già specificate.

- **Gestione dei certificati**: precedentemente menzionata, la tecnica di comunicazione dominante tra microservizi è mTLS, se non si dispone l'uso di intermediari (ESB, Pub-Sub, Proxy Mesh). Ciò significa che è necessario utilizzare i certificati per rendere le comunicazioni sicure. Se da un lato si possono respingere facilmente comunicazioni provenienti dall'esterno, dall'interno si presentano gli stessi problemi di scarsa flessibilità dei certificati. I meccanismi di revoca, in particolare, sono lenti in termini di propagazione dell'informazione. Inoltre, abbiamo già menzionato i problemi nella scelta di avere una CA esterna o interna.
- **Autorizzazione – il problema del "Confused Deputy"**: problema definito da Norm Hardy (46), per cui un utente, tramite un programma, può accedere risorse per cui non ha autorizzazione tramite un secondo processo o servizio privilegiato, che agisce senza sapere se l'utente abbia o meno i requisiti di autorizzazione per farlo. Di base è un problema di delega "logica" e di fiducia: il processo privilegiato che agisce si fida del fatto che la richiesta sia lecita, perché arriva da un componente e l'identità dietro la richiesta è quella del componente, non del reale utente. Nel caso del sistema a microservizi, questo è un reale problema perché i servizi spesso lavorano in "catena". Una richiesta passa da un servizio ad un altro. Normalmente sarebbe un problema di design di sistema, spesso dovuto al fatto di avere autorità diverse e conflittuali e di implementare con leggerezza il sistema di autorizzazione per delega, ma questo dettaglio va considerato quando si cerca di evidenziare criticità e priorità nelle difese di un servizio.

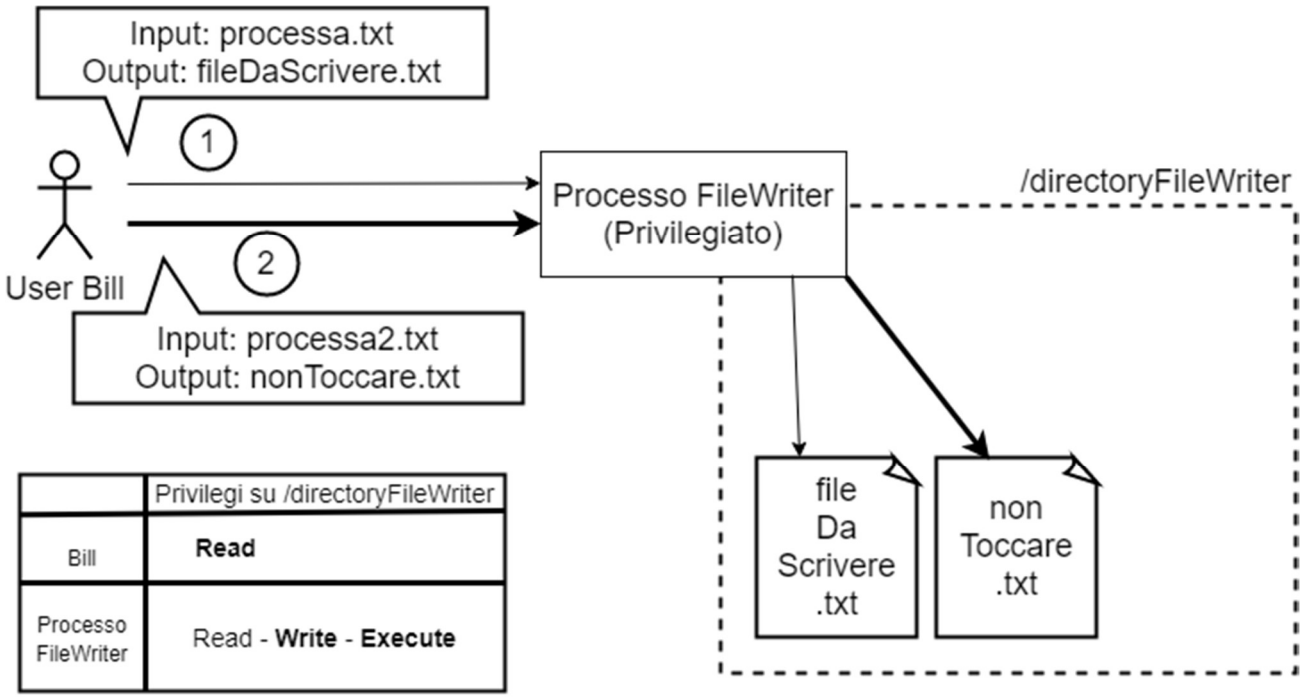

Figura 15 - Schematizzazione semplificata del problema Confused Deputy. Implicitamente, il file con nome "nonToccare.txt" non dovrebbe essere modificato, ma, la logica e i permessi del Processo FileWriter, che detiene il "controllo" della directory per evitare modifiche da parte degli utenti, gli permettono di scrivere sul file da non toccare.

Queste due problematiche sono relativamente nuove nell'ambito della sicurezza informatica. Mentre il problema della gestione dei certificati è meno fresca, il problema del vice confuso (che è anche definizione di un particolare attacco informatico) assume una nuova forma, per cui è necessario sempre di più avere un'idea molto precisa della topologia e del flusso di controllo/esecuzione del sistema.

Considerando lo schema e i punti precedenti, quindi è evidente individuare tre elementi su cui andranno concentrati gli sforzi:

- Il servizio "Utenti", a cui il Gateway accede direttamente
- Il Service Registry a cui tutti i servizi si registrano, per essere visibili e per contattarsi a vicenda, e che il Gateway interroga per individuare il servizio a cui indirizzare richieste.
- Il servizio "Accounts", con priorità lievemente minore rispetto ai precedenti, ma di vitale importanza in quanto permette di "manipolare" il dominio dell'applicazione: Accounts fornisce informazioni ad Users e trasferisce le informazioni sulle transazioni a Transactions, di fatto "validandole".

Da questi tre punti, abbiamo tre considerazioni importanti:

- La topologia di un'app a microservizi influenza di molto le valutazioni strategiche in materia di difesa. Una buona architettura applicativa, come in questo caso, diminuisce i punti di accesso, canalizzandoli verso due possibili obiettivi e isolando il resto dal contatto diretto verso l'esterno. Poter discernere facilmente quali sono i servizi più facilmente raggiungibili e quali meno raggiungibili permette di impostare meglio il discorso di priorità verso determinate parti del sistema.
- L'interconnessione tra i vari servizi, a livello di flusso d'esecuzione, permette di individuare immediatamente dei percorsi che l'attaccante potrebbe voler fare lateralmente: le frecce in grassetto in Figura 10 individuano il flusso operativo, individuando anche una possibile "strada" per l'attaccante tra i vari servizi. Se un'attaccante volesse ottenere informazioni bancarie sull'utente, non potendole ottenere da "Users", dovrebbe prima attraversare "Users" per poi infiltrarsi in "Accounts".

La dipendenza tra i vari servizi, a livello di contesto, permette di comprendere come indirizzare gli sforzi di sicurezza. Ad esempio, Transactions, in teoria, è solo un deposito di "dati". Contiene e mantiene integri i registri di transazioni. L'unica operazione eventualmente possibile è la lettura di queste transizioni e l'aggiunta di nuove transizioni, non la scrittura. Mentre Accounts contiene le informazioni bancarie del singolo utente, quindi informazioni sensibili e attivabili (a scopo di furto materiale o d'identità), Transactions contiene informazioni statiche. Tuttavia Transactions contiene le transazioni, che permettono di ricostruire lo storico di un utente, controllano quini la coerenza dei dati contenuti in Account e tracciandone quindi i movimenti bancari. Quindi se Accounts richiederà una difesa più corposa, l'obiettivo del Blue Team su Transactions sarà la tutela dell'integrità (e della riservatezza) delle informazioni ivi contenute.

La superficie d'attacco internamente si restringe quindi ai servizi "frontali" e ai canali di comunicazione. L'unico modo in cui la penetrazione o l'infezione può proseguire è spostandosi attraverso i canali di comunicazione, ma il presupposto è la conquista di uno dei due servizi. In questo senso, possiamo individuare quindi una sorta di schema delle possibili minacce e dei possibili vettori d'attacco.

3.7.1. Topologia e comunicazione

Da quanto scritto precedentemente, emerge che la comunicazione inter-server non è differente da quella user-to-service, ad eccezione dei dettagli precedentemente elencati. Ciò significa che possiamo rifarci tranquillamente alle tassonomie tradizionali, includendo però le seguenti considerazioni:

- Ogni comunicazione proveniente dall'esterno deve essere bloccata a priori. I servizi comunicano solo tra loro e l'unico accesso ammesso "dall'esterno" è l'API Gateway.
- Esistono precisi percorsi e direzioni di comunicazione, una volta mappato il sistema adeguatamente, permettendo di eseguire ancor meglio le operazioni di filtraggio e di indirizzare meglio gli alert.
- Una corretta mappatura permette di valutare l'anatomia dell'applicazione, in modo da distribuire gli sforzi difensivi in maniera intelligente. Comprendere compiti e risorse di ogni servizio permette di individuare facilmente cosa controllare.
- I servizi comunicano tra loro attraverso il container. Alcuni servizi possono essere malevoli: bisogna sicuramente capire come distinguere un servizio malevolo da uno lecito e bloccare qualsiasi tentativo di comunicazione.
- E' essenziale vigilare sull'autorizzazione effettiva a disposizione di un utente, quando richiede operazioni ed esse si propagano nel sistema, per impedire effetti indesiderati od operazioni illecite.

Se dovessimo fare un esempio pratico, proviamo ad osservare la differenza concettuale tra Users, Accounts e Transactions e come questi comunicano: possiamo elaborare regole di whitelisting per i canali di comunicazione e capire come intervenire a difesa di un servizio. Concentrandoci su Transactions:

- Transactions comunica con il Service Registry e Account. Ciò significa che qualunque packet filter dovrà impedire comunicazioni diverse da quelle evidenziate. Se si pone anche un monitor sul traffico, questo dovrà immediatamente generare un alert nel momento in cui arriva traffico da qualsiasi entità che non siano quelle elencate precedentemente (whitelisting).
- Transactions, concettualmente, si occupa solo di raccogliere, ufficializzare e documentare transazioni. Quando Accounts conclude le operazioni di transazione, vengono comunicate a Transactions e registrate. Viceversa, Account potrebbe richiedere a Transactions le transazioni di uno specifico account bancario. Ciò delimita le possibili operazioni che Accounts può richiedere a Transaction: Accounts ad esempio non può chiedere a Transactions di eliminare transazioni. L'obiettivo quindi è bloccare qualsiasi tentativo di operare in maniera anomala e soprattutto garantire l'integrità delle informazioni contenute in Transaction.

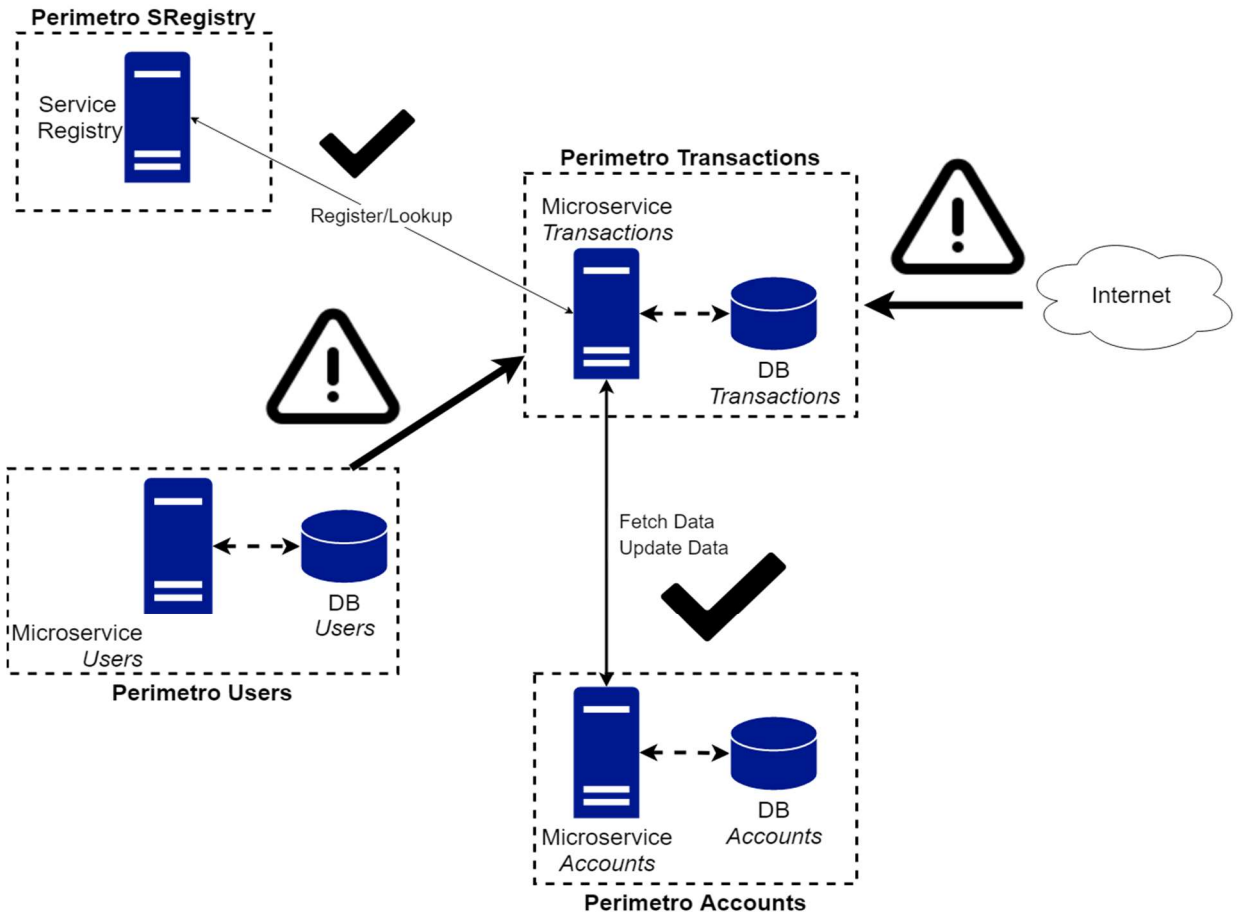

Figura 16 - L'esempio precedente di Transaction. Qualsiasi cosa non arrivi da Accounts o dal Service Registry deve essere considerata un'anomalia, perché il sistema è pensato in quel modo. Sembra una considerazione "rigida" da fare, ma permette in realtà di poter discriminare subito connessioni potenzialmente dannose, invece di dover accettare tutte le connessioni e cercare la singola connessione pericolosa.

Un altro scenario immediatamente anomalo è rappresentato da un servizio che effettua port scanning o ping sweep (Nmap è uno strumento, ad esempio, che esegue proprio queste operazioni): questo evento deve essere catturato immediatamente come un'anomalia, all'interno della rete. La motivazione è ovvia: i servizi vengono messi in comunicazione tra loro tramite il servizio di discovery, provvederà poi il servizio di discovery a indirizzare un servizio verso il suo specifico tramite, fornendo i dettagli di localizzazione per la connessione. Come anche le comunicazioni tra servizi avvengono, in termini di funzionamento, solo tramite mTLS. Qualsiasi altro protocollo (ICMP, SSH, …) proveniente da uno qualsiasi degli host degli altri servizi deve immediatamente mettere in allerta, in quanto c'è una probabilità molto alta che sia un'azione effettuata da un intruso.

La logica rimane sempre la stessa: se un servizio comunica attraverso canali e modalità diverse da quelle utilizzate, questo è controllato da un'entità terza non appartenente al sistema stesso. Quindi un attaccante, che sia interno od esterno. Per il resto, l'insieme delle tecniche e dei vettori d'attacco rimane lo stesso di una qualsiasi rete o di una web application, in quanto cambia il contesto di comunicazione, non come questa viene effettivamente portata avanti.

<table>
<tr><th>Superficie d'attacco</th><th>Vettori d'attacco</th><th>Minacce / Attacchi</th></tr>
<tr><td>Network (generale)</td><td>Connessioni non-mTLS
Connessioni esterne
Token d'accesso modificati
API non sanitizzate</td><td rowspan="2">Injection
Broken authentication
Sensitive data exposure
Broken access control (Confused-deputy attack)
Insecure deserialization
DDoS
Elevation of privilege
Spoofing
Tampering</td></tr>
<tr><td>Network (strutturale)</td><td>Servizi malevoli
Service registry compromesso
CA compromessa</td></tr>
</table>

Tabella 3 - Possibile tassonomia di sicurezza per la comunicazione intra-servizi in ambito Microservizi.

3.7.2. Esempio di Remote Elevation of Privilege: SSH e container.

Un esempio di remote elevation of privilege è fornito dal writeup indicato nel capitolo "Esempio di Container Escape" come fonte per i possibili attacchi più complessi. Il writeup fa riferimento ad una challenge data durante uno dei workshop di sicurezza di Docker. Nonostante sia una challenge, quindi sia fornito un sistema "vulnerabile", permette di vedere gli effetti della vulnerabilità e di un possibile attacco.

Citando direttamente il writeup (43), la challenge consiste in quanto segue: "I partecipanti avranno accesso SSH ad un server remoto in AWS. L'obiettivo è mostrare che un attaccante può eseguire un processo come utente root in un altro server nella rete locale, avviando un servizio Docker non sicuro".

Tramite l'accesso SSH, l'attaccante effettua le operazioni di mapping dal server intermedio per individuare il server bersaglio e verificare se ha un servizio Docker. Individuato il servizio di Docker esposto, prova a connettersi usando il client sul server intermedio, ma il servizio non può essere raggiunto. L'attaccante decide quindi di creare un'immagine non sicura di un container, in modo da usare il client di Docker del server intermedio per effettuare l'upload dell'immagine sul servizio Docker.

Una volta effettuato l'upload, dal server intermedio, si può avviare il container: nel farlo, lo si avvia come utente "root" (quindi permessi massimi). Questo permette di montare il filesystem (la directory radice "/") dentro la directory "/mnt" del container. A questo punto, l'attaccante può accede a due diversi obiettivi:

- **authorized_keys (./ssh)**: accede alla directory "./ssh" dell'utente "ubuntu" per inserire la propria chiave SSH tra le chiavi autorizzate per l'utente ubuntu. L'attaccante può fare SSH verso il server remoto senza bisogno di password.
- **sudoers**: può accedere al file di accesso ai privilegi "root" (tramite comando sudo) per fare in modo che l'utente "ubuntu" (un utente semplice) possa usare "sudo" senza password.

Il container malevolo quindi ha modificato tutto ciò che è necessario per permettere di effettuare una connessione SSH remota al server bersaglio senza necessitare di alcuna password (né per SSH, né per "sudo") ed eseguire codice arbitrario.

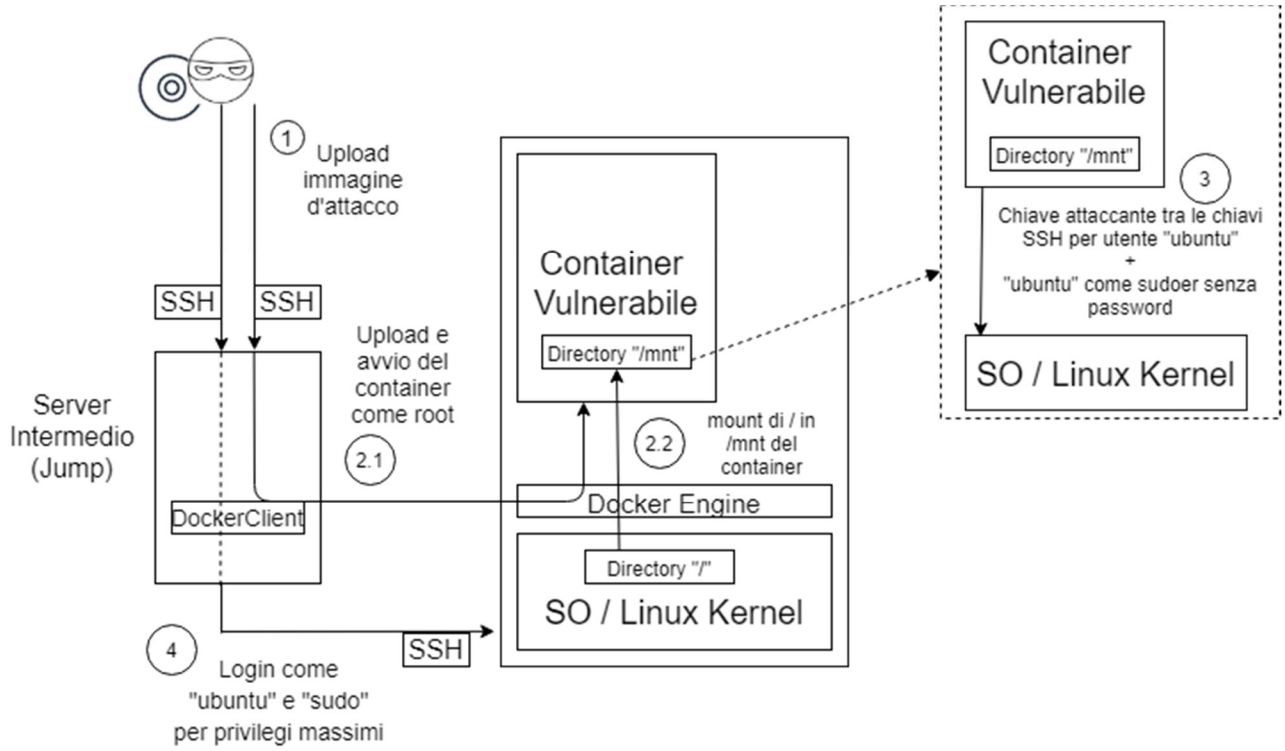

Figura 17 - Schematizzazione dell'attacco di Remote Elevation of Privilege: da notare, sul server intermedio, come la connessione SSH sia condizione necessaria per l'attacco, dato che tutto passa attraverso il server intermedio. L'attacco in sé mostra come connessioni non controllate possono portare ad un attacco laterale, portando l'attaccante nel sistema senza che l'attaccante interagisca direttamente con il sistema remoto.

Questo writeup dimostra gli effetti devastanti di permettere connessioni dall'esterno presso un servizio. Sebbene la sfida non trattasse di un sistema a microservizi, writeup di questo tipo evidenziano la gravità di permettere connessioni esterne, specie se non vengono impostati settaggi di sicurezza forti e non viene controllato l'assegnamento e il possesso di determinati privilegi, sia da parte dei container che degli utenti all'interno del SO ospitante.

3.7.3. Difendersi nell'ambiente intra-server

Abbiamo già accennato all'utilizzo estensivo di IDS, packet filter/sniffer e file integrity checker, strumenti comunque validi dato che, sebbene cambi il contesto (e quindi gli attori) nella comunicazione, sempre di una comunicazione di rete si tratta. La gestione dei container precedentemente menzionata è uno strumento molto comodo per rispondere immediatamente ad un attaccante all'interno del sistema.

In questo senso, vorrei mettere in evidenza, del writeup precedentemente menzionato, il processo iniziale. Nonostante la descrizione riferisse chiaramente come il bersaglio fosse un secondo server, nessun dato è stato fornito per indicare dove il server si trovasse, forzando l'attaccante a dover effettuare operazioni di discovery e mapping. Il movimento laterale dell'attaccante è possibile solo in presenza di informazioni, che in genere vengono raccolte in diverse fasi di ricognizione del sistema bersaglio. Collegandoci al discorso della topologia le operazioni di ricognizione, in un sistema a microservizi come l'esempio usato per quest'analisi, possono fare molto rumore, soprattutto le scansioni della rete. Il vantaggio è che queste possono essere immediatamente identificate e possono essere predisposte azioni preventive: in prima istanza si può reagire immediatamente nel bloccare l'attaccante in ricognizione, impedendogli di prelevare maggiori informazioni. Successivamente, data l'estrema flessibilità dei container, è possibile cambiare contenitori per cambiare la mappa della rete interna del sistema a microservizi, in modo che indirizzi e punti d'accesso non rimangano statici, vanificando le operazioni di ricognizione e costringendo l'attaccante a ricominciare da capo il procedimento.In presenza di strumenti che permettano quindi di avere una visione e un controllo globali delle interazioni, specie di rete, la particolare struttura di un'architettura a microservizi può rappresentare un vantaggio strategico enorme.

Di notevole importanza è la PoC di Yarygina circa un possibile attacco low-level nei confronti di un'applicazione a microservizi contro la stessa applicazione strutturata secondo il modello monolitico. Nella PoC, Yarygina vuole mettere in evidenza i vantaggi intrinsechi dell'architettura a Microservizi.

L'applicazione MicroBank, in due versioni, viene posta sotto attacco per poter eseguire azioni arbitrarie, per gonfiare il conto corrente dell'attaccante. Si assume che l'attaccante abbia una copia del codice: l'obiettivo dell'attaccante è quello di usare un buffer overflow per accedere alle aree di codice contenenti "gadgets", cioè pezzi di codice contenuti nell'applicazione che possono essere usati a vantaggio dell'attaccante. Una volta ottenuto, l'attaccante può eseguire operazioni arbitrarie con i privilegi della banca stessa.

A differenza della versione monolitica dello scenario, nella versione microservizi l'exploit non riesce completamente nell'obiettivo.

Si conclude infatti che:

- L'API Gateway deve essere sfruttato per accedere. L'attacco non può manipolare quindi le risorse, perché sono contenute in un database che non è più presente nella stessa macchina del punto d'accesso. Questo comporta che l'attaccante deve sfruttare prima l'API Gateway per convincere uno dei microservizi a "lavorare per lui". Questo è possibile perché, nel normale stato di esecuzione, i servizi interni si fidano solo dell'API Gateway per ricevere comandi.
- Il servizio potrebbe non avere lo stesso set di "gadgets" utilizzati per l'exploit, per cui è necessario riadattare il codice dello script. La PoC ha come ipotesi che l'attaccante conosca il codice, ma in realtà è molto difficile che l'attaccante abbia una conoscenza così profonda del sistema, rendendo questo tipo di attacco molto difficile. Allo stesso modo, dato che i servizi sono "piccoli" e quindi hanno una base di codice minore, si assottigliano le possibilità di trovare gadget utili per eseguire l'exploit.
- Se si usasse il concetto di diversificazione, sarebbe ancora più difficile per l'attaccante riuscire ad ottenere una reazione a catena tale da poter effettuare un attacco, in quanto richiederebbe più di uno script, adattato al servizio bersaglio. Ancor più se la diversificazione è data dal diverso linguaggio in cui sono scritti i servizi stessi.
- Aumentando la difficoltà, le tempistiche e la complessità necessaria per uno spostamento laterale, si rendono gli IDS molto più efficaci per identificare e prevenire l'attacco. Inoltre, nonostante sia possibile non far scattare gli IDS, "il difensore non perde il flusso di controllo di tutto il sistema e ha possibilità di mitigare questo tipo di attacchi".

Questo rappresenta un vantaggio enorme in termini difensivi, perché rende più macchinosi gli attacchi, dando ai difensori non solo più tempo per reagire ad un attacco, ma permette di rendere più "omogenei" i pesi delle conseguenze di un attacco: se si riflette sui ruoli, si evidenzia immediatamente una disparità in termini di rischi e ricompensa. Un attaccante può attaccare più volte, ma allo stato attuale un attacco comporta la ricompensa totale. Nonostante il primo attacco possa fallire, qualora l'attaccante sia stato sufficientemente cauto da permettere di non essere tracciato, l'attaccante può anche ritentare all'infinito: l'attaccante può sempre "iniziare la partita" e ritentare, mentre la sconfitta, per il difensore, ha un peso molto più alto, rispetto alla ricompensa di un attaccante per la riuscita dell'attacco.

Dalla PoC di Yarygina emerge che il difensore può diminuire questo modo il peso di un attacco riuscito: l'attacco, per riuscire completamente, deve essere portato lateralmente attraverso N nodi. Anche se l'attaccante riesca a penetrare, non è detto che riesca ad ottenere risorse considerabili preziose immediatamente: il team di difesa può "permettersi" un'intrusione, anzi, ha un margine di tempo più grande per identificare l'intruso prima che possa effettivamente accedere con successo a delle risorse.

Tuttavia mancano soluzioni, in forma di framework o Intrusion Response Systems, che abbiano la stessa portata e lo stesso impatto del processo d'orchestrazione in termini di sicurezza, a sostegno di una versione globale, centralizzata e generale, mentre gli strumenti difensivi localizzati funzionano ancora perfettamente. Infatti Yarygina sfrutta la PoC per proporre un IRS automatizzato, sfruttando l'intelligenza artificiale e modellando lo scenario di un attacco come un "gioco a somma zero finito e dinamico, a due giocatori" (44) [11] L'obiettivo di questo IRS è appunto l'automatizzazione dei meccanismi difensivi, in favore di una maggiore velocità di risposta ed efficienza.

[11] Cap. 6.2.1., p. 99

3.8. Risultato finale: una strategia multi-livello

Per sintetizzare quanto rilevato analizzando la struttura ideale in esame, è possibile tradurre una possibile formula per le operazioni di sicurezza informatica nell'ambito dei Microservizi:

1. **Mappatura dell'applicazione**: prima di procedere occorre elaborare una mappa della topologia della rete, in cui è presente ogni servizio incluso nell'applicazione. Nella mappa devono essere ben in evidenza le funzionalità dei servizi, l'obiettivo di ogni servizio, le interazioni tra i servizi e le direzioni di comunicazione.
2. **Valutazione strategica**: una volta ottenuta la mappatura, si procede con la stima di priorità, in termini di sicurezza, dei servizi. Per ogni servizio, si valutano le criticità in termini di logica di business e di interazioni con altri servizi e si assegna una priorità in base ai criteri di sicurezza che quel servizio specifico deve avere, oltre a valutare possibili vulnerabilità.
3. **Fortificazione:** si preparano i sistemi difensivi per la struttura. Si divide in:
   3.1. **Fortificazione servizi**: si procede ad armare i singoli servizi con i meccanismi di sicurezza specifici per il servizio, costruendo un perimetro difensivo adatto allo stesso. Si procede quindi a vietare qualsiasi connessione dall'esterno, ad attivare il monitoraggio delle connessioni lecite, ad attivare gli strumenti di salvaguardia dell'integrità per i file critici, soprattutto legati alle configurazioni di creazione di container e accesso, e per le immagini container da utilizzare in futuro.
   3.2. **Fortificazione di rete**: in base al ruolo e alle interazioni tra i servizi, si procede ad applicare strumenti per il monitoraggio del traffico e per la verifica dei servizi stessi.
4. **Tracciamento fortificazioni**: alla mappatura vengono aggiunte le soluzioni difensive applicate ai singoli servizi e alla rete stessa, in modo da avere una visione organica dell'intera applicazione dal punto di vista della sicurezza. Per rete stessa, si intende l'atto di mettere in evidenza le direttrici coperte dalle soluzioni difensive in termini di interazioni, in modo da tenere traccia delle comunicazioni poste sotto sorveglianza e protezione.

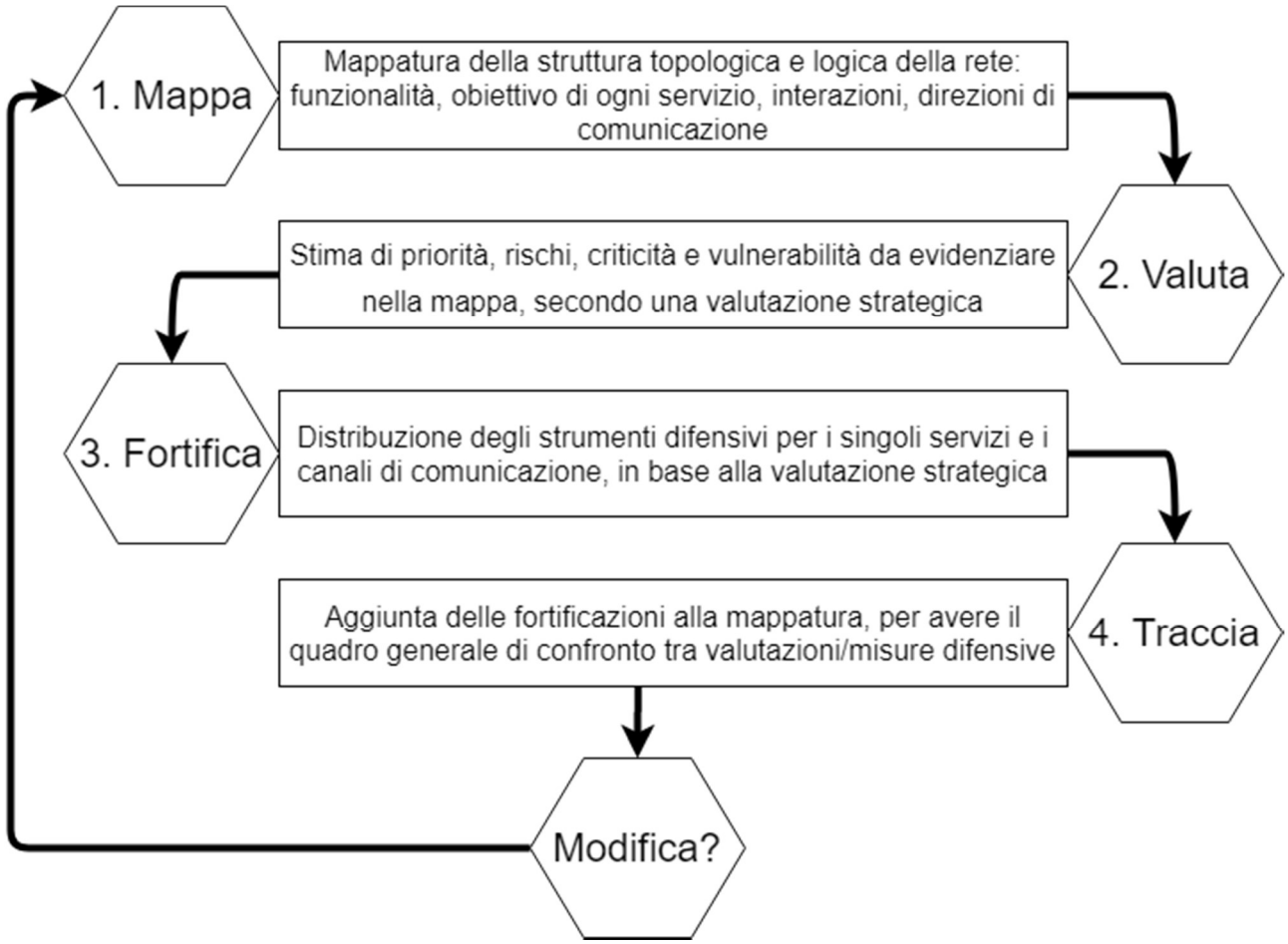

Figura 18 – Schematizzazione del possibile processo strategico da intraprendere. I quattro step principali sono ciclici, considerando anche la modifica del sistema. E' implicito che la modifica del sistema non comporterà di ricominciare da zero, ma di integrare le modifiche con la mappa precedentemente ottenuta.

La strategia delineata e rappresentata in Figura 18 permette di orientarsi nel processo di realizzazione delle strategie particolari per un Blue Team. I passi elencati sono intesi come linee guida, da adattare in base alla situazione. Seguendo quindi il flusso definito in questa tesi, è possibile immaginare il lavoro di ogni passo in questo modo:

1. La mappatura è il processo effettuato dal capitolo 3.3 al capitolo 3.5, con risultato la Figura 10 e la Figura 11. Si procede ad ottenere una rappresentazione organica dell'applicazione e di come questa funziona.
2. La valutazione è il processo effettuato al capitolo 3.4.1, 3.4.2 e 3.5. Si ridefiniscono i limiti del sistema, si comprende come questo lavora e si mettono in evidenza tutte le informazioni utili in termini di sicurezza.
3. La fortificazione è idealmente rappresentata dai capitoli 3.6 e 3.7. Sebbene non ci siano delle nette indicazioni difensive, come il setup di un IDS, la pulizia specifica delle configurazioni del sistema di containerizzazione in termini di permessi e di policy di capabilities, la preparazione del comparto di logging, la logica del discorso rimane quella di partire dal singolo server e risalire bottom-up la struttura, seguendo le indicazioni ottenute al passo di valutazione.

4. Il tracciamento altro non è che l'aggiunta di ogni misura di fortificazione alla mappa precedentemente delineata. Si parte dalla Figura 11 e si aggiungono i dettagli degli strumenti utilizzati per la difesa, sia per i singoli host che per la comunicazione tra servizi.
5. Inoltre, i passi di mappatura e aggiornamento non sono dei meri passi di raccolta e aggregazione di informazioni: supponiamo che un nuovo servizio venga aggiunto. La flessibilità di un sistema a microservizi ritorna estremamente utile: non è necessario ricominciare da capo con una nuova mappatura, ma aggiornare la mappa precedentemente fatta, applicando i passi di "Valutazione strategica" sia al servizio nuovo che ai servizi che interagiranno con il nuovo servizio, applicando i passi di "Fortificazione" al nuovo servizio e aggiornando ove necessario.

3.9. Conclusioni

Il lavoro di ricerca fin qui esposto permette di trarre conclusioni circa le domande poste all'inizio di questa tesi.

1) Quanto è veramente estesa la superficie d'attacco in un sistema a Microservizi?

La superficie d'attacco in un sistema a Microservizi è sicuramente più ristretta, grazie alla struttura con cui possono essere sviluppate applicazioni tramite Microservizi. Tuttavia, possibili implementazioni o topologie scorrette possono aumentarla, rendendo molto più complesso riuscire a difendere il sistema.

Nonostante l'utilizzo di più tecnologie e anche di linguaggi differenti, rispetto ad un tradizionale sistema distribuito con applicazioni ad architettura monolitica, i vantaggi precedentemente esposti permettono di controbilanciare questo aspetto garantendo più resistenza a spostamenti laterali, rendendo in alcuni scenari più complesso, per un attaccante, portare a termine un attacco contro un servizio specifico.

2) È possibile, a partire da un modello di architettura a Microservizi e ricordando quanto già esiste per l'architettura monolitica, individuare una strategia difensiva applicabile a tutta l'architettura (Blue team strategy)?

Sì, è decisamente possibile individuare una strategia generale applicabile a tutta l'architettura. Soprattutto il concetto di Defense-in-depth può essere tradotto in una base di partenza nell'imbastimento delle specifiche strategie, perché fornisce un punto di partenza con l'analisi del sistema e delle sue meccaniche interne. Certamente, ogni strategia va riadattata alla diversa struttura dei microservizi.

Un approccio seguirebbe quindi la valutazione delle tecnologie dello stato dell'arte usate nell'applicazione o nella struttura per poi eseguire un'analisi bottom up, partendo dai singoli host e dalle possibili vulnerabilità e concludendo con la rete interna, per tracciare una mappa strategica chiara, volta ad una distribuzione efficace delle difese.

3) Sono presenti strumenti efficaci per le operazioni di individuazione, contenimento ed espulsione e stima di danni in caso di un avversario già all'interno di un componente della nostra architettura?

Sì, gli strumenti già presenti di sicurezza menzionati, soprattutto IDS e IPS, funzionano ancora più che egregiamente nell'architettura a Microservizi. Come già evidenziato, la differenza sostanziale tra i modelli precedenti e i microservizi è l'utilizzo di tecnologie nuove come la containerizzazione e i cloud e la struttura di rete interna, che è identica ad una rete esterna, ma in cui le interazioni tra i componenti sono note a priori.

Per la prima differenza, gli strumenti a garanzia dell'integrità e di gestione di autorizzazione sono estremamente efficaci, mentre per la seconda gli strumenti di difesa per il networking funzionano meglio, perché si hanno già le informazioni per farne una messa a punto più precisa, rispetto al caso del networking proveniente dalla rete esterna.

L'unico inconveniente strumentale è l'assenza di framework o soluzioni armoniche che permettano di coordinare le difese su tutti gli elementi dell'architettura in maniera omogenea e rapida, ma ci sono già ricerche attive in questo senso. Tuttavia, la presenza di strategie generali e ben schematizzate, come nel capitolo 3.8, potrebbe risultare fondamentale per la costruzione di strumenti per la difesa organizzata di interi sistemi.

4) E' possibile ricavare quantomeno considerazioni generali, da applicare a possibili modelli difensivi per architetture a Microservizi? Esistono possibili soluzioni?

Come menzionato nella risposta alla domanda 1, esistono considerazioni generali da applicare a modelli difensivi, anche se non esiste una soluzione "unica" d'uso generale. Come detto, il modello difensivo ha determinate considerazioni generali che poi vanno applicate al caso in esame (alla struttura dell'applicazione che un Blue Team andrà a difendere). Considerare a priori come dannose connessioni provenienti dall'esterno è un esempio di considerazioni generali, a cui bisogna applicare, in caso, eccezioni strutturali. Un esempio è la presenza di macchine che si connettono dall'esterno per eseguire azioni di amministrazione, che debbono essere trattate comunque come un caso particolare.

Un problema maggiore, in questo senso, è la necessità di coordinazione tra il personale IT di sviluppo e il team di difesa: un canale di comunicazione continuo permette di tenere aggiornata la mappa dell'applicazione e quindi di rispondere rapidamente a cambiamenti nella stessa, senza perdere di vista il controllo globale che si può avere. Non solo, è anche importante capire il contesto di determinati servizi, per gestire meglio la distribuzione degli sforzi difensivi, e per questo è necessario che i team di sviluppo forniscano informazioni circa la logica di un servizio nella struttura.

Bibliografia

1. NGINX, Inc. The Future of Application Development and Delivery is Now. [Online] Novembre 2015. https://www.nginxcom/resources/library/app-dev-survey/.

2. Research and Markets. https://www.researchandmarkets.com/research/szk939/microservice. [Online] Settembre 2017.

3. *Overcoming Security Challenges in Microservice Architectures.* T. Yarygina, A. H. Bagge. 12th IEEE Symposium on Service-Oriented System Engineering : SOSE 2018, 2018. pp. Cap. 2.1.1, Definition 2, p. 19.

4. https://aws.amacon.com/it/message/41926/. *Summary of the Amazon S3 Service Disruption in the Northern Virginia (US-EAST-1) Region.* [Online]

5. Riccio, K. Can Your Company Endure an AWS-size Outage? *DataCenter Knowledge.* [Online] 03 13, 2017. https::://www.datacenterknowledge.com/archives/2017/03/13/can-company-endure-aws-size-outage.

6. Damon, E. dev2ops. *What is DevOps?* [Online] 02 22, 2010. https://web.archive.org/web/20120909013310/http://dev2ops.org/blog/2010/2/22/what-is-devops.html.

7. SOA Manifesto. [Online] 2009. http://www.soa-manifesto.org/.

8. Bell, M. *Introduction to Service-Oriented Modeling. Service-Oriented Modeling: Service Analysis, Design and Architecture.* s.l. : Wiley & Sons., 2008. ISBN: 978-0-470-14111-3.

9. Zimmerman, O. *Microservices tenets: Agile approach to service development and deployment.* s.l. : Computer Science - Research and Development, 2016. ISSN: 1865-2024.

10. *Microservices: The Journey So Far and Challenges Ahead*". P. Jamshidi, C. Pahl, N. C. Mendonça, J. Lewis, S. Tilkov. 3, s.l. : IEEE Software, pp. 24-35, 2018, Vol. 35. DOI: 10.1109/MS.2018.2141039.

11. Microsoft. AppContainer Isolation. *Microsoft Docs.* [Online] 05 31, 2018. https://docs.microsoft.com/it-it/windows/win32/secauthz/appcontainer-isolation?redirectedfrom=MSDN.

12. Docker. What is a Container? *Docker.* [Online] https://www.docker.com/resources/what-container.

13. Dahan, U. Bus and Broker Pub/Sub Differences. *Udi Dahan - The Software Simplist.* [Online] 03 24, 2011. https://udidahan.com/2011/03/24/bus-and-broker-pubsub-differences/.

14. Schneider, F. B. CS 513 System Security -- Something You Know, Have or Are. *Cornell CIS Computer Science.* [Online] 2005. http://www.cs.cornell.edu/courses/cs513/2005fa/NNLauthPeople.html.

15. Google, Harris Poll. *Google.* [Online] Febbraio 2019. https://www.incibe-cert.es/en/blog/password-based-authentication.

16. Cimpanu, C. 44 million Microsoft users reused passwords in the first three months of 2019. *ZDNet.* [Online] Dicembre 5, 2019. https://www.zdnet.com/article/44-million-microsoft-users-reused-passwords-in-the-first-three-months-of-2019/.

17. Federal Financial Institutions Examination Council - FFIEC. FFIEC Press Release - October 12, 2005. *FFIEC.* [Online] Ottobre 12, 2005. https://www.ffiec.gov/press/pr101205.htm.

18. Brook, C. NIST Recommends SMS Two-Factor Authentication Deprecation. *threat post.* [Online] Luglio 27, 2016. https://www.zdnet.com/article/nist-blog-clarifies-sms-deprecation-in-wake-of-media-tailspin/.

19. Fontana, J. NIST blog clarifies SMS deprecation in wake of media tailspin. *ZDNet.* [Online] Luglio 29, 2016. https://www.zdnet.com/article/nist-blog-clarifies-sms-deprecation-in-wake-of-media-tailspin/.

20. NIST. NIST Special Publication 800-63B. *nist.gov.* [Online] Gennaio 29, 2021. https://pages.nist.gov/800-63-3/sp800-63b.html.

21. Google. Google Workspace Updates: New settings for 2-Step Verification. *googleblog.com.* [Online] Giugno 20, 2016. https://workspaceupdates.googleblog.com/2016/06/new-settings-for-2-step-verification.html.

22. Miller, C. Apple prompting iOS 10.3 users to enable two-factor authentication w/ push notification. *9to5mac.com.* [Online] Febbraio 25, 2017. https://9to5mac.com/2017/02/25/two-factor-authentication-ios-10-3/.

23. Direttiva (UE) 2015/2366 del Parlamento europeo e del Consiglio del 25 novembre 2015 relativa ai servizi di pagamento nel mercato interno, che modifica le direttive 2002/65/CE, 2009/110/CE e 2013/36/UE e il regolamento (UE) n. 1093/2010. *EUR-Lex.* [Online] Dicembre 23, 2015. https://eur-lex.europa.eu/eli/dir/2015/2366/oj/ita.

24. AltroConsumo. Token digitale, le banche dicono addio alla chiavetta. *Sito web AltroConsumo.* [Online] Giugno 10, 2019. https://www.altroconsumo.it/soldi/conti-correnti/news/token-digitale.

25. IEFT. RFC 7519 - JSON Web Token (JWT). *IEFT Tools.* [Online] Maggio 2015. https://tools.ietf.org/html/rfc7519.

26. J. Manico, J. Maćkowski. HTML5 Security - OWASP Cheat Sheet Series. *cheatsheetseries.owasp.org.* [Online] https://cheatsheetseries.owasp.org/cheatsheets/HTML5_Security_Cheat_Sheet.html#local-storage.

27. Siriwardena, P. Securing Microservices (Part I). *medium.facilelogin.com.* [Online] Aprile 12, 2016. https://medium.facilelogin.com/securing-microservices-with-oauth-2-0-jwt-and-xacml-d03770a9a838.

28. IEFT. RFC 6749 - The OAuth 2.0 Authorization Framework. *IEFT Tools.* [Online] Ottobre 2012. https://tools.ietf.org/html/rfc6749.

29. OpenID. OpenID Connect | OpenID. [Online] https://openid.net/connect/.

30. IEFT. RFC 5264 - The Transport Layer Security (TLS) Protocol Version 1.2. *IEFT Tools.* [Online] Agosto 2008. https://tools.ietf.org/html/rfc5246.

31. —. RFC 5878 - Transport Layer Security (TLS) Authorization Extensions. *IEFT Tools.* [Online] Maggio 2010. https://tools.ietf.org/html/rfc5878.

32. McGuiness, T. Defence-in-Depth. *SANS Institute Reading Room.* [Online] Novembre 11, 2001. https://www.sans.org/reading-room/whitepapers/basics/defense-in-depth-525.

33. National Security Agency - NSA. Defense in Depth. *NSA IAD.gov Library Archive.* [Online] Marzo 12, 2010. https://apps.nsa.gov/iaarchive/library/ia-guidance/archive/defense-in-depth.cfm.

34. *Low-Level Exploitation Mitigation by Diverse Microservices.* C. Otterstad, T. Yarygina. s.l. : F. De Paoli, S. Schulte, E. Broch Johnsen, 2017, Service-Oriented and Cloud Computin (ESSOC 2017).

35. Suomalainen, J. Defense-in-Depth Methods in Microservices Access Control. s.l. : Tampere University, Febbraio 2019.

36. Team, CSP-CERT Blue. Advisories - Meltdown and Spectre | Cyber Security Philippines. *Cybersecurity Philippines CERT (cert.ph).* [Online] Luglio 2018. https://www.cert.ph/advisories/2018/2018-07-meltdown-and-spectre.html.

37. *Developing Dependable and Secure Cloud Application.* I. Weber, S. Nepal, L. Zhu. 3, s.l. : IEEE Internet Computing, Maggio-Giugno 2016, Vol. 20, pp. 74-79.

38. Lietz, S. What is DevSecOps? - DevSecOps. *devsecops.org.* [Online] Giugno 1, 2015. https://www.devsecops.org/blog/2015/2/15/what-is-devsecops.

39. *Guarding the castle keep: teaching with the fortress metaphor.* D. A. Frincke, M. Bishop. 3, Maggio-Giugno 2004, IEEE Security & Privacy, Vol. 2, pp. 69-72.

40. *Beyond the Castle Model of cyber-risk and cyber-security.* C. Leuprecht, et al. 2016, Government Information Quarterly.

41. *Virtualization vulnerabilities, security issues, and solutions: a critical study and comparison.* D. Tank, A. Aggarwal, N. Chaubey. s.l. : International Journal of Information Technology, Febbraio 2019.

42. Czarnota, D. Understanding Docker container escapes. *Trail of Bits Blog (blog.trailofbits.com).* [Online] Luglio 19, 2019. https://blog.trailofbits.com/2019/07/19/understanding-docker-container-escapes/.

43. "ch0ks", A. Puente Z. Hacking Docker Remotely | H4CKarandas. *H4CKarandas (hackarandas.com).* [Online] Marzo 17, 2020. https://hackarandas.com/blog/2020/03/17/hacking-docker-remotely/.

44. *A Game of Microservices: Automated Intrusion Response.* T. Yarygina, C. Otterstad. [ed.] E. Rivière S. Bonomi. s.l. : Springer International Publishing, Giugno 2018, Distributed Applications and Interoperable Systems, pp. 169-177.

45. *N-VERSION PROGRAMMINC: A FAULT-TOLERANCE APPROACH TO RELIABILITY OF SOFTWARE OPERATION.* L. Chen, A. Avizienis. s.l. : IEEE, 1995. Twenty-Fifth International Symposium on Fault-Tolerant Computing, 1995, ' Highlights from Twenty-Five Years'. ISBN: 0-8186-7150-5.

46. Hady, N. The Confused Deputy. [Online] Dicembre 05, 2003. https://web.archive.org/web/20031205034929/http:/www.cis.upenn.edu/~KeyKOS/ConfusedDeputy.html.