IMPLEMENTASI *MUTUAL TRANSPORT LAYER SECURITY* (mTLS) PADA ARSITEKTUR MICRO*SERVICE*S DENGAN ISTIO DI KUBERNETES

Disusun Oleh:

N a m a : Riski Wahyu Kurniawan

NIM : 15523069

PROGRAM STUDI TEKNIK INFORMATIKA – PROGRAM SARJANA

FAKULTAS TEKNOLOGI INDUSTRI

UNIVERSITAS ISLAM INDONESIA

2020

HALAMAN PENGESAHAN DOSEN PEMBIMBING

IMPLEMENTASI *MUTUAL TRANSPORT LAYER SECURITY* (mTLS) PADA ARSITEKTUR MICRO*SERVICE*S DENGAN ISTIO DI KUBERNETES

TUGAS AKHIR

Disusun Oleh:

N a m a	: Riski Wahyu Kurniawan
NIM	: 15523069

Yogyakarta, 22 Juli 2020

Pembimbing,

(Fietyata Yudha, S.Kom., M.Kom)

HALAMAN PENGESAHAN DOSEN PENGUJI

IMPLEMENTASI *MUTUAL TRANSPORT LAYER SECURITY* (mTLS) PADA ARSITEKTUR MICRO*SERVICE*S DENGAN ISTIO DI KUBERNETES

TUGAS AKHIR

Telah dipertahankan di depan sidang penguji sebagai salah satu syarat untuk memperoleh gelar Sarjana Komputer dari Program Studi Teknik Informatika di Fakultas Teknologi Industri Universitas Islam Indonesia

Yogyakarta, 27 Agustus 2020

Tim Penguji

Fietyata Yudha, S.Kom., M.Kom. _____

Anggota 1

Mukhammad Andri Setiawan, S.T., M.Sc., ph.D. _____

Anggota 2

Ari Sujarwo, S.Kom., M.I.T. _____

Mengetahui,

Ketua Program Studi Teknik Informatika – Program Sarjana

Fakultas Teknologi Industri

Universitas Islam Indonesia

(Dr. Raden Teduh Dirgahayu, S.T., M.Sc.)

HALAMAN PERNYATAAN KEASLIAN TUGAS AKHIR

Yang bertanda tangan di bawah ini:

Nama : Riski Wahyu Kurniawan

NIM : 15523069

Tugas akhir dengan judul:

IMPLEMENTASI *MUTUAL TRANSPORT LAYER SECURITY* (mTLS) PADA ARSITEKTUR MICRO*SERVICE*S DENGAN ISTIO DI KUBERNETES

Menyatakan bahwa seluruh komponen dan isi dalam tugas akhir ini adalah hasil karya saya sendiri. Apabila dikemudian hari terbukti ada beberapa bagian dari karya ini adalah bukan hasil karya sendiri, tugas akhir yang diajukan sebagai hasil karya sendiri ini siap ditarik kembali dan siap menanggung resiko dan konsekuensi apapun.

Demikian surat pernyataan ini dibuat, semoga dapat dipergunakan sebagaimana mestinya.

Yogyakarta, 22 Juli 2020

(Riski Wahyu Kurniawan)

HALAMAN PERSEMBAHAN

Teruntuk Ibu yang telah merawat dan mendidik anaknya dan telah sabar menghadapi ambisi dan kenakalan anaknya, serta selalu memanjatkan do'a di setiap waktu untuk keberhasilan anaknya

HALAMAN MOTO

"Menyepi itu penting, supaya kamu benar-benar bisa mendengar apa yang menjadi isi dari keramaian" **—***Emha Ainun Nadjib*

KATA PENGANTAR

Assalamu'alaikum Warahmatullahi Wabarakatuh

Puji syukur penulis panjatkan atas kehadirat Allah SWT karena berkat rahmat taufik serta karunia Nya sehingga penulis dapat menyelesaikan laporan tugas akhir berjudul **"Implementasi *mutual Transport Layer Security* (mTLS) Pada Arsitektur Micro*service*s dengan Istio di Kubernetes"** ini dapat terselesaikan sebagaimana mestinya.

Tugas akhir ini dibuat sebagai salah satu syarat yang harus dipenuhi untuk menyelesaikan pendidikan jenjang sarjana dari Program Studi Teknik Informatika, Fakultas Teknologi Industri, Universitas Islam Indonesia.

Dalam proses penyelesaian tugas akhir ini, tidak sedikit rintangan dan hambatan yang harus dilalui oleh penulis. Namun, berbagai rintangan dan hambatan tersebut akhirnya dapat dilewati dikarenakan dukungan dari berbagai pihak. Maka atas hal tersebut, penulis ingin mengucapkan terima kasih kepada:

Kata pengantar adalah bagian yang digunakan untuk menyampaikan rasa syukur atas selesainya penyusunan laporan tugas akhir. Selain itu, bagian kata pengantar juga dapat membuat berbagai hal sebagai berikut:

1. Allah SWT yang telah memberikan banyak petunjuk serta kelancaran untuk menyelesaikan tugas akhir ini.
2. Mahmudatin Rufi'ah selaku Ibu dari penulis yang senantiasa mendo'akan tanpa putus di tiap waktu ibadahnya demi kelancaran akademik anaknya.
3. Septyan Prima Prayoga selaku kakak kandung penulis yang selalu mendukung dan memberikan semangat.
4. Bapak Fietyata Yudha, S.Kom., M.Kom. selaku dosen pembimbing yang berkenan membimbing dan memberikan saran serta masukan untuk penulis.
5. Sahabat-sahabat yang menjadi tempat diskusi dan inspirasi ngulik dunia IT, Ryan "pntek", Mas Galang, Ryan "django", Alvin Miftah, Ade Bagas, Hilman Maulana.
6. Sahabat-sahabat yang ada di kontrakan "Zoo" Degolan yang senantiasa mendukung, menginspirasi, dan bersedia menjadi tempat untuk bercerita dan berdiskusi.
7. Teman-teman yang membantu proses penulisan laporan, Faishal dan Kartika.
8. Semua karyawan di Bridestory yang memberikan ide dan inspirasi judul.
9. Seluruh teman dan pihak-pihak yang telah membantu kelancaran proses penelitian ini yang tidak dapat disebutkan satu persatu.

Selain ucapan terima kasih, permohonan maaf tak lupa penulis sampaikan kepada para pembaca laporan ini apabila terdapat kekurangan. Dengan selesainya penulisan laporan tugas akhir ini, besar harapan penulis agar karya ini dapat memberikan manfaat dan dapat dikembangkan di masa yang akan datang.

Yogyakarta, 22 Juli 2020

(Riski Wahyu Kurniawan)

SARI

Beberapa tahun belakangan ini tren arsitektur *microservices* menjadi sorotan dan perbincangan di dunia *developing*. *Microservices* sendiri merupakan metode pengembangan aplikasi menjadi beberapa bagian rangkaian aplikasi kecil dan menjadikanya sebuah *services* sendiri. Selain itu perbincangan tentang kejadian pembocoran dan pelanggaran data mengalami kenaikan sebesar 126% dibandingkan tahun sebelumnya. Salah satu cara mengamankan komunikasi antar *services* pada *microservices* adalah dengan *mutual Transport Layer Security* (mTLS) dengan memanfaatkan fitur keamanan dari *Service Mesh* dan salah satu tools tersebut adalah Istio yang berjalan diatas infrastruktur Kubernetes. Tujuan dari penelitian adalah mengetahui bagaimana implementasi kemudian cara kerja dan perbedaan komunikasi antara yang tidak dan menggunakan mTLS dan hasil perbandingan performa dari *microservices* tersebut dengan bantuan beberapa tools yang sudah ada sehingga dapat membantu pihak *developer* maupun *operation* untuk memperhatikan ranah keamanan dalam aplikasi yang sedang dikerjakan.

Kata kunci: *mutual Transport Layer Security*, *Microservices*, *Service Mesh*, Kubernetes

GLOSARIUM

DNS	Singkatan dari *Domain Name System* yang menjadi sistem penamaan komputer, layanan atau sumber daya yang terkoneksi dengan suatu jaringan
Service(s)	Hasil dari kumpulan perangkat lunak atau aplikasi sebagai kumpulan layanan
Cluster	Pengelompokkan sumber daya
Container	Sistem yang terisolasi
Protocol	Standar yang digunakan untuk terjadinya komunikasi di jaringan
Request	Permintaan komunikasi jaringan dengan suatu protokol
Load Balancer	Sistem yang digunakan untuk mendistribusikan trafik jaringan
Sidecar Container	Sistem tambahan yang terisolasi dan mendukung kontainer utama
Tools	Sistem atau aplikasi yang digunakan untuk pengawakutuan sistem lainya
Server	Sistem komputer yang menyediakan jenis layanan tertentu
Client	Orang atau sistem yang menggunakan sumber daya dari server
Stdout	Singkatan dari *Standard Output* yang menjadi deskriptor aplikasi untuk keluaran

DAFTAR ISI

DAFTAR TABEL

DAFTAR GAMBAR

BAB I
PENDAHULUAN

1.1 Latar Belakang

Seiring kebutuhan proses bisnis yang berjalan di suatu perusahaan atau instansi, menimbulkan banyak rancangan atau ide untuk mengimprovisasikan pada proses bisnis tersebut. Salah satu cara untuk membantu improvisasi proses bisnis tersebut di bidang Teknologi Informasi adalah dengan pemilihan metode arsitektur pengembangan aplikasi. Menurut Sasa Baskarada, et al (2018: 9), Beberapa tahun belakangan ini tren arsitektur pengembangan dengan *microservice* mengalami peningkatan yang signifikan ditunjukkan dari hasil Google Trends semenjak tahun 2015. Sedangkan untuk pengertian *microservices* sendiri yaitu arsitektur metode pengembangan aplikasi menjadi beberapa bagian rangkaian aplikasi kecil dan menjadikanya sebuah *services* sendiri dimana *services* tersebut berjalan sendiri dari proses bisnis, pengembangan sampai berjalan secara independen (Indrasiri & Siriwardena, 2018)

Semakin tinggi kompleksitas dari proses bisnis yang berjalan, membuat semakin rumit juga dalam pengembangan aplikasi dengan metode *microservices*. Mulai dari bagaimana cara memanajemeni dalam hal *deployment*, *monitoring* dan keamanan untuk banyak aplikasi yang berjalan dengan arsitektur *microservices*. Bicara soal dunia keamanan, terjadi banyak kejahatan *data leakage* (pembocoran data) dan *data breach* (pelanggaran data) yang terjadi di dunia bisnis. Menurut laporan dari "Identity Theft Resource Center" pada tahun 2018 kejadian *data breaches exposed* naik sekitar 126% dibandingkan tahun sebelumnya. Karena kompleksitas dalam manajemen micro*service*s dan ditambah masalah keamanan yang serius salah satu caranya adalah dengan metode *service mesh* (Indrasiri & Siriwardena, 2018). *Service mesh* merupakan metode yang bertujuan sebagai jembatan antara setiap *services* dan memberinya sebuah kemampuan seperti manajemen *traffic*, *load balancing*, *resilience*, *observability* dan *security* (Sutter, 2018)

Dari uraian fitur yang diberikan oleh *service mesh* di atas, salah satu *tool* untuk mendukung *service mesh* adalah Istio dan fitur *security* memiliki peranan penting dalam proses manajemen *microservices* karena peretas bisa saja dapat melihat ataupun mungkin dapat mencuri informasi di dalam proses komunikasi antar *microservices* di dalam *service mesh*. Pada Istio bagian keamanan memberikan fitur untuk melakukan *mutual authentication*

atau autentikasi dua arah pada setiap *service* yang ada di dalam *service mesh*. Setiap komunikasi *service* akan diberikan *certificate* baik yang berperan sebagai *server* dan *client* akan saling bertukar menukar *certificate* untuk autentikasinya. Proses autentikasi bertukar menukar tersebut disebut dengan *mutual Transport Layer Security* (mTLS) *authentication*.

Salah satu syarat untuk menjalankan Istio adalah harus di atas infrastruktur Kubernetes. Kubernetes merupakan salah satu alat untuk *orchestration container* atau orkestrasi kontainer secara *clustering*. Maksud dari orkestrasi adalah untuk melakukan manajemen sebuah kontainer secara otomatis mulai dari proses *computing*, *deployment*, penjadwalan atau *scheduling* sampai dengan manajemen untuk bagian *networking* dan untuk menjalankan sebuah aplikasi atau *services* di dalam Kubernetes harus dalam bentuk kontainer.

Untuk itu dalam penelitian ini, penulis akan membuat bagaimana cara membangun komunikasi secara aman pada micro*service*s dengan *mutual Transport Layer Security* (mTLS) pada arsitektur *service mesh* dengan Kubernetes dan melakukan pengujian untuk mengetahui perbedaan antara aplikasi yang tidak menggunakan mTLS dan yang menggunakanya serta dengan pengujian performa dari *microservices* tersebut. Penelitian ini diharapkan dapat membantu pihak *developer* dan *operation* untuk lebih memperhatikan ranah keamanan dalam aplikasi atau *services* yang mereka kerjakan.

1.2 Rumusan Masalah

Berdasarkan latar belakang yang sudah dipaparkan, berikut adalah rumusan masalah dari penelitian yang akan dilakukan:

a. Bagaimana cara membangun komunikasi antar micro*service*s dengan *mutual Transport Layer Security* (mTLS) menggunakan Istio?
b. Bagaimana cara mengetahui komunikasi micro*service*s yang tidak menggunakan *mutual Transport Layer Security* (mTLS) dengan yang sudah menggunakan mTLS?
c. Bagaimana perbandingan performa *microservices* yang tidak menggunakan *mutual Transport Layer Security* (mTLS) dengan yang sudah menggunakan mTLS?

1.3 Tujuan Penelitian

Adapun tujuan dari penelitian ini diantaranya:

a. Mengetahui cara komunikasi dengan *mutual Transport Layer Security* (mTLS) pada micro*service*s dengan Istio.

b. Mengetahui cara pengujian komunikasi micro*service*s yang menggunakan *mutual Transport Layer Security* (mTLS) dengan yang tidak menggunakan mTLS.
c. Mengetahui perbandingan performa micro*service*s yang menggunakan *mutual Transport Layer Security* (mTLS) dengan yang tidak menggunakan mTLS.

1.4 Batasan Penelitian

Terdapat beberapa batasan masalah yang diambil untuk membatasi sasaran utama dalam tugas akhir ini adalah sebagai berikut:

a. Versi Istio yang digunakan adalah versi 1.4.9.
b. Fitur keamanan pada Istio yang digunakan hanya *mutual Transport Layer Security* (mTLS).
c. Aplikasi *microservices* yang digunakan dalam pengujian hanya dua *microservices* berupa satu aplikasi *frontend* dan 1 *aplikasi backend*.

1.5 Manfaat Penelitian

Adapun manfaat yang diharapkan dari penelitian ini antara lain:

a. Mengetahui bagaimana cara autentikasi dua arah atau *mutual Transport Layer Security* (mTLS) dapat diterapkan untuk komunikasi antar *microservices*.
b. Memberikan gambaran tentang pentingnya faktor keamanan ketika melakukan proses *developing* pada arsitektur *microservices.*

1.6 Sistematika Penulisan

a. **BAB I PENDAHULUAN**

   Bab ini berisi uraian tentang latar belakang, rumusan masalah, tujuan penelitian, batasan masalah, manfaat penelitian, dan sistematika penulisan.

b. **BAB II LANDASAN TEORI**

   Bab ini berisi uraian tentang penelitian sebelumnya yang terkait dengan penelitian yang dilakukan saat ini serta dasar teori dalam melakukan implementasi *mutual Transport Layer Security* pada arsitektur *microservices*.

c. **BAB III METODOLOGI PENELITIAN**

   Bab ini berisi uraian tentang langkah-langkah yang telah dilakukan selama penelitian.

d. **BAB IV HASIL DAN PEMBAHASAN**

Bab ini berisi uraian tentang hasil dari implementasi *mutual Transport Layer Security* pada *microservices* beserta dengan pengujianya.

e. **BAB V KESIMPULAN DAN SARAN**

Bab ini berisi kesimpulan akhir dari penelitian dan saran yang kiranya dapat dipertimbangkan untuk pengembangan dan perbaikan dalam penelitian selanjutnya.

BAB II
LANDASAN TEORI

2.1 Penelitian Sebelumnya

Bagian ini akan menjelaskan terkait penelitian-penelitian yang telah dilakukan sebelumnya untuk kemudian dijadikan acuan dalam proses penelitian ini. Tabel 2.1 menunjukkan daftar penelitian sebelumnya yang menjadi dasar dalam penelitian ini.

Tabel 2.1 Penelitian sebelumnya

No	Judul	Penulis	Deskripsi	Saran
1	*Overcoming Security Challenges in Microservice Architectures*	(Yarygina & Bagge, 2018)	Pada penelitian ini membahas tentang taksonomi dari *microservices* kemudian menganalisa masalah keamanan dari *level layer microservices* kemudian memberikan hasil perbandingan kinerja antara konsep yang dibuat dengan menerapkan keamanan pada *microservices* dan yang tidak menerapkan konsep keamanan pada *microservices.*	Seiring pertumbuhan industri yang mengadopsi arsitektur *microservices*, harus ditunjang juga dengan pertumbuhan dunia penelitian untuk mengamankan arsitektur *microservices* dan kedepannya solusi tentang *security* dapat mudah digunakan, *developer side*, dan juga ringan untuk menunjang performa.
2	*Protected Coordination of Service Mesh for Container-based 3-tier Service*	(Kang et al., 2019)	Pada penelitian ini membahas tentang pemanfaatan *service mesh* dengan Istio untuk mempermudah	Konsep *container-based 3-tier traffic* ini dapat melakukan proses enkripsi dan dekripsi untuk

	Traffic		manajemen dari sisi operasional dan membuat skema untuk mengatur *traffic* dan enkripsi komunikasi data yang ada di *microservices.*	beberapa pembagian traffic yang berbeda dan memberikan hasil yang bagus pada *stress test*.
3	*Security in Microservices Architectures*	(Coelho, 2018)	Pada penelitian ini memaparkan beberapa resiko celah keamanan pada arsitektur *microservices* dan memberikan beberapa cara penanganannya.	Mempelajari dan mengimplementasikan konsep dan metode pengamanan untuk sekarang adalah salah satu hal wajib dalam proses pengembangan aplikasi dengan arsitektur *microservices*.
4	*Defense-in-depth Methods in Microservices Access Control*	(Suomalainen, 2019)	Pada penelitian ini memiliki tujuan untuk melakukan evaluasi dan perbandingan analisa dari desain arsitektur *microservices* dengan prinsip keamanan yaitu *confidentiality*, *authenticity*, dan *integrity*.	Penelitian ini berharap untuk penelitian selanjutnya adalah dilakukan analisa lebih dalam tentang komunikasi antar *services* dan lebih berfokus kepada keamanan *container orchestration*.
5	*Service Isolation in Large Microservice Networks*	(Palm, 2018)	Pada penelitian ini berfokus pada beberapa tahapan pengujian yaitu: 1. Pengujian *Service*	Istio merupakan salah satu produk yang menarik dan memiliki bentuk modular dan menyediakan kemampuan untuk

			Isolation. 2. *Service Modification*. 3. *Gradual Rollout*. 4. *Management access to protected services*. 5. *High availability*.	memanajemeni *microservices* dengan metode *service mesh*.

Persamaan dari penelitian di atas adalah sama-sama membahas dan menggunakan metode *mutual Transport Layer Security* (mTLS) sebagai salah satu bentuk mengamankan komunikasi antar *services* pada arsitektur *microservices*. Untuk penelitian ini akan dilakukan penerapan atau implementasi sampai dengan tahap uji coba lebih dalam untuk metode *mutual Transport Layer Security* (mTLS) pada arsitektur *microservices* dengan beberapa *tools* yang ada.

2.2 Kajian Literatur

Pada bagian ini akan menjelaskan hasil dari penelitian sebelumnya yang akan dijadikan sebagai pokok bahasan sudut pandang sebelum masuk ke dalam dasar teori.

2.2.1 Autentikasi

Dari beberapa uraian dari penelitian sebelumnya, penulis mengambil pokok bahasan tentang autentikasi yang menjadi salah satu syarat utama dalam *security control* keamanan pada arsitektur *microservices* (Coelho, 2018). Autentikasi sendiri adalah proses untuk mengidentifikasi sebuah pengguna, sistem, *services* maupun sesuatu yang merupakan unik (Siriwardena, 2020). Dalam hal ini, tingkatan atau level autentikasi *microservices* biasanya terjadi di dua bagian, yang pertama merupakan autentikasi di level *end user* atau pengguna dan yang kedua terjadi pada level *services*.

Metode yang dapat diterapkan pada level autentikasi antar *services* adalah metode dengan menggunakan *token* dan menggunakan *mutual Transport Layer Security* (Indrasiri & Siriwardena, 2018). Penggunaan *token* ini biasanya disebut juga dengan *JSON Web Token*

(JWT) sehingga informasi yang dapat digunakan sebagai tanda identifikasinya adalah hasil dari *generate token* dengan standar dari rfc 7519 (Jones et al., 2015). Sedangkan untuk *mutual Transport Layer Security* (mTLS) sendiri menggunakan pertukaran tanda identifikasi dengan standar sertifikat dengan memanfaatkan protokol SSL/TLS.

2.2.2 *Service Mesh*

Dengan kompleksitas pengembangan arsitektur *microservices* dan dengan banyaknya jumlah *services* yang harus dimanajemeni mulai dari bagaimana antar *services* berkomunikasi satu sama lain, pemantauan setiap *services* hingga dengan sisi keamanan antar *services*. Salah satu *tools* yang dapat membantu menyelesaikan kompleksitas tersebut adalah *Service Mesh*. *Service Mesh* memberikan fungsi aturan pada level *networking* pengembangan arsitektur *microservices* dan memberinya kemampuan seperti *traffic control*, *load balancing*, *observability* dan kemampuan keamanan (Calcote, 2018).

Pemberian kemampuan tersebut secara teknis menerapkan *sidecar pattern* yaitu sebuah komponen aplikasi yang terletak bersamaan dengan *services* utamanya (Indrasiri & Siriwardena, 2018). Dalam layer *service mesh*, terdapat dua layer yaitu *data plane* dan *control plane* kemudian *sidecar pattern* yang sudah disebutkan sebelumnya adalah termasuk bagian dari *data plane* (Wolff, E., & Prinz, 2020). Selanjutnya untuk level tertinggi yang bertugas mengatur banyak *data plane* adalah *control plane* yang memberikan sekumpulan aturan dan konfigurasi pada *Service Mesh*. Berikut merupakan beberapa produk tools *Service Mesh*, yaitu:

a. Istio
b. Consul
c. Linkerd

2.2.3 Penggunaan

Penggunaan dua metode autentikasi di tingkat antar *services* sering kita temui di pengembangan arsitektur *microservices* seperti layanan kesehatan, komunikasi antar *services* di tingkat penyedia *cloud computing* dan autentikasi microservices *Internet of Things* (IoT) *devices* (Alkhulaifi & El-Alfy, 2020). Penggunaan metode autentikasi mTLS biasanya terjadi pada dunia perbankan dan salah satu contohnya adalah VUB (developers@vub.sk, 2019) dan contoh penggunaan kedua metode autentikasi tersebut di dunia ekonomi dan perbankan dari WSO2 (WSO2 Inc, 2020).

Selain contoh penggunaan autentikasi dari dua metode tersebut, selanjutnya adalah contoh penggunaan umum *Service Mesh* adalah membantu melakukan improvisasi pemantauan atau *monitoring* dari *microservices* dengan adanya fitur *tracing* sehingga memudahkan dalam proses *troubleshooting*. Kemudian penggunaan fitur trafik control pada proses pengembangan *microservices* dan pemanfaatan fitur keamanan seperti automasi JWT dan mTLS (Manpathak, 2019).

2.2.4 Biaya

Seiring bertambahnya tingkat keamanan dan metode keamanan yang digunakan, membuat bertambahnya *cost* atau biaya yang dibutuhkan (The Economist Newspaper, 2015). Penggunaan kedua metode tersebut membutuhkan biaya pemrosesan sumber daya dan komputasi untuk melakukan pekerjaan keamananya. Penggunaan JWT dengan pihak ketiga dikenakan biaya setiap request antar aplikasinya. Sedangkan penggunaan mTLS memiliki biaya yang lebih tinggi dari spesifikasi sumber daya komputasi yang dibutuhkan, biaya pemeliharaan dan juga aspek *high-availability* (Siriwardena, 2020).

Dari biaya yang dibutuhkan untuk *maintaining* dan *operating* sebuah *Service Mesh*, laporan awal tahun 2019 tentang implementasi dan pemanfaatan *Service Mesh* dapat menekan biaya operasi seiring dengan kompleksitas dan kebutuhan *services* (Garrett, 2019). Namun untuk menjalankan *Service Mesh* sendiri membutuhkan sumber daya komputasi lebih sehingga mempengaruhi dalam biaya operasional.

Dari uraian pokok empat bahasan sudut pandang diatas, penulis dapat menarik kesimpulan sementara bahwa autentikasi merupakan salah satu faktor utama dalam proses pengamanan dalam komunikasi antar services pada arsitektur microservices (Monahan, 2019). Kemudian sekitar 81% terdapat serangan pada tingkat API antar services (Groskop, M., Pariente, N., Scialabba, L. and Smith, 2020). Pemilihan metode atau cara Service Mesh menjadi salah satu jalan untuk mempermudah manajemen autentikasi dengan fitur automasi pada tingkat sertifikat untuk mutual Transport Layer Security (mTLS).

2.3 Dasar Teori

2.3.1 *Mutual Transport Layer Security*

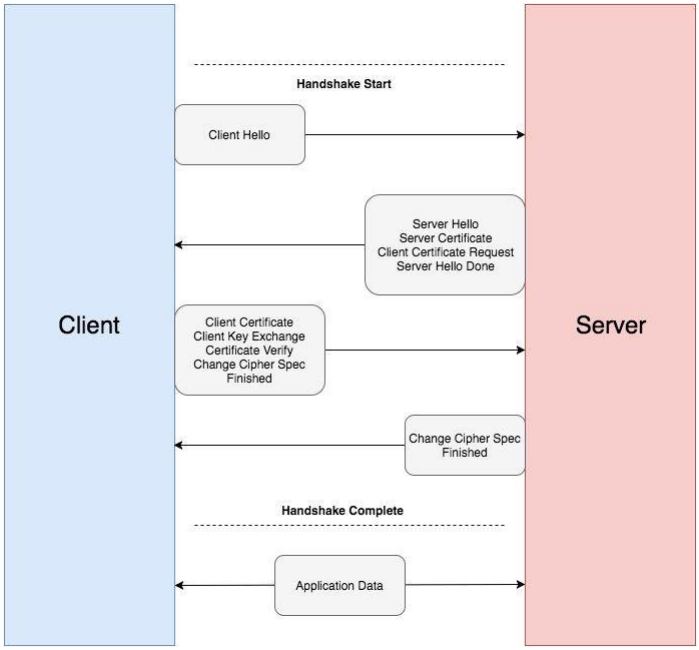
Transport Layer Security (TLS) merupakan salah satu protokol yang menyediakan proses autentikasi untuk keamanan antara komunikasi jaringan. *Transport Layer Security* (TLS) merupakan pengembangan dari teknologi *Secure Socket Layer* (SSL) yang dalam penggunaanya menggunakan algoritma pertukaran kunci Diffie-Hellman. Protokol SSL/TLS ini memfasilitasi penggunaan enkripsi untuk data yang rahasia dan membantu menjamin integritas informasi yang dipertukarkan antara *client* dan *server* (Munir, 2019).

Dari pengertian tentang protokol SSL/TLS diatas, Proses dari metode *mutual Transport Layer Security* ini membutuhkan *mutual authentication* yaitu proses autentikasi yang berjalan dua arah dari sisi *client* dan *server*. Proses autentikasi dua arah ini secara standar menggunakan x.509 *Certificate* yaitu sebuah sertifikat digital yang ada pada kunci publik. Sertifikat digital adalah dokumen elektronis yang digunakan untuk membuktikan kepemilikan kunci publik. Sertifikat digital ini dikeluarkan oleh pemegang *Certification Authority* (CA). Pada x.509 Certificate mempunyai beberapa komponen utama di dalam isi sertifikatnya. Tabel 2.2 menjelaskan komponen utama yang ada di x.509 Certificate.

Tabel 2.2 Komponen utama x.509 Certificate

Isi	**Arti**
Version	Versi untuk x.509 Certificate.
Serial Number	Nomor yang digunakan untuk mengidentifikasi sertifikat.
Signature Algorithm	Algoritma yang digunakan untuk digital signature.
Issuer	Nama untuk pemegang Certification Authority (CA).
Validity Period	Waktu untuk berlakunya sertifikat.
Subject Name	Berisi entitas yang disertifikasi.
Subject Public Key	Algoritma kriptografi yang digunakan untuk public key.
Issuer ID	Opsional untuk ID pemegang Certification Authority.
Subject ID	Opsional untuk ID yang disertifikasi.
Extensions	Opsional untuk ekstensi tambahan.
Signature	Tanda tangan digital yang dimiliki oleh Issuer.
Signature Algorithm	Algoritma yang digunakan untuk tanda tangan digital.

Untuk menjalankan sertifikasi digital seperti x.509 Certificate, dibutuhkan sebuah infrastruktur untuk memberikan fasilitas pengiriman dan pertukaran sertifikat secara aman dan nama infrastruktur tersebut adalah *Public Key Infrastructure* (PKI). PKI merupakan kumpulan beberapa aturan yang terdiri dari kebijakan untuk membuat, mendistribusikan, menggunakan, menyimpan, mengolah dan membuang atau mengganti sertifikat digital (Munir, 2019). Gambar 2.1 merupakan gambar visualisasi *handshake* mutual Trans*port* Layer Security (mTLS).



Gambar 2.1 *Handshake* mTLS

2.3.2 Micro*service*s

Microservices adalah bentuk arsitektur metode pengembangan aplikasi menjadi beberapa bagian rangkaian aplikasi kecil dan menjadikanya sebuah *services* sendiri di mana *services* tersebut berjalan sendiri dari proses proses bisnis dan dalam pengembanganya berjalan secara independen (Indrasiri & Siriwardena, 2018). Selain dari hal pengertian, menurut Yarygina Tetiana dan Bagge Anya H, *microservices* mempunyai beberapa karakteristik khusus yaitu:

a. *Distributed composition*

   Microservices dibangun di atas *microservices* yang lain dan berkomunikasi dengan menggunakan teknologi jaringan yang ada.

b. *Modularity*

Microservices bekerja dengan API dan mudah untuk digunakan kembali beberapa modulnya di *microservices* yang lain.

c. *Encapsulation*

Setiap *services* yang ada pada *microservices* harus berada dalam keadaan terenkapsulasi.

d. *Network service*

Services harus mudah dijangkau dan mudah di akses di atas jaringan.

Untuk aspek keamanan komunikasi antar *service* pada *microservices* menggunakan sistem *Authentication* (autentikasi) dan *Authorization* (otorisasi) sama seperti pada salah satu keamanan pada tingkat pengguna atau *user*. Menurut sumber dari buku *Building Microservices* karangan Sam Newman, terdapat beberapa pilihan dalam mengamankan komunikasi antar tiap *services* pada *microservices* di antaranya, yaitu:

a. Perimeter

Semua komunikasi *service* harus berada dalam lingkungan atau tempat yang aman dan biasanya cara ini dilakukan pada level keamanan jaringan.

b. HTTP(S) Authentication

Menerapkan standar autentikasi pada setiap *services* menggunakan protokol http (web) yang sudah memiliki *SSL Certificates*.

c. Client Certificate

Setiap *services* di dalam arsitektur *microservices* diberikan kemampuan untuk berkomunikasi di tingkat *Transport Layer Security* (TLS) dan setiap *client* dalam hal ini *services* atau aplikasi diberikan *X.509 Certificate* yang akan digunakan sebagai bentuk verifikasi kuat dan valid.

d. HMAC Over HTTP

Hash-based Messaging code (HMAC) merupakan salah satu optional untuk komunikasi *services* pada arsitektur *microservices* dengan cara memberikan tambahan kode *hash* untuk setiap proses *request* autentikasinya. Tambahan kode *hash* tersebut disimpan dan dikirim melalui protokol http.

2.3.3 Istio

Istio merupakan sebuah alat yang dapat berjalan di atas Kubernetes dan mempunyai fungsi untuk melakukan *Service Mesh.* merupakan metode yang bertujuan sebagai jembatan antara setiap *services* dan memberinya sebuah kemampuan seperti manajemen *traffic*, *load*

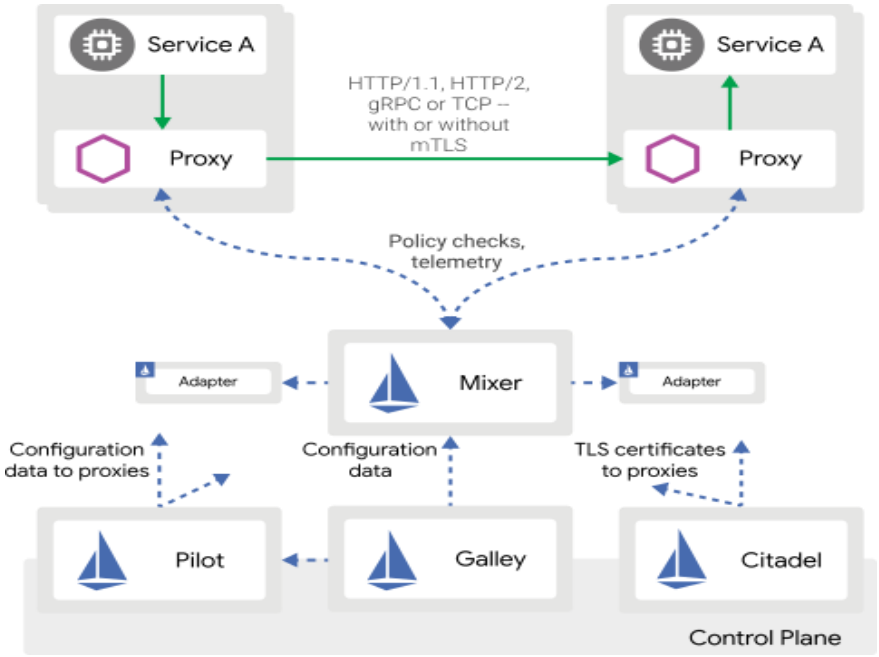
balancing, *resilience*, *observability* dan *security* (Sutter, 2018). Secara *logical*, Istio menggunakan prinsip seperti cara kerja *router* yaitu terbagi menjadi dua *plane*, yaitu:

a. Data Plane

Di dalam Istio mempunyai tugas sebagai pengatur *proxy* yang ditempelkan ke kontainer dalam bentuk sidecars. *Proxy* tersebut mempunyai fungsi untuk mengatur dan menjembatani semua komunikasi jaringan di dalam *microservices*.

b. Control Plane

Control Plane yang ada di dalam Istio mempunyai fungsi untuk melakukan konfigurasi routing ke dalam *proxy* dan juga mengatur akses untuk tiap *microservices* (Calcote & Butcher, 2020). Gambar 2.2 menunjukkan arsitektur dan komponen dari Istio.



Gambar 2.2 Arsitektur dan komponen pada Istio (Istio Authors, 2019)

Dari Gambar 2.2, dapat diketahui bahwa Istio memiliki beberapa pembagian plane lagi yaitu:

a. Envoy

Envoy merupakan *proxy server* yang memiliki tingkat keandalan yang tinggi dan dikembangkan dengan bahasa C++. Tugas utama envoy adalah menjembatani semua komunikasi data yang masuk dan keluar untuk semua *Service* di tiap *microservices* (Calcote & Butcher, 2020).

b. Mixer

Merupakan salah satu komponen dari Istio yang berdiri sendiri. Mixer berfungsi untuk memberikan *access control* dan pengaturan akses dari komunikasi *Service mesh*.

c. Pilot

Pilot di dalam Istio memberikan layanan untuk *Service discovery*. *Service discovery* merupakan sebuah metode untuk melakukan pengecekan secara berkala terhadap setiap *Service* yang ada dalam suatu sistem.

d. Citadel

Citadel adalah salah satu komponen yang ada pada Istio dan bertugas untuk melakukan pengaturan keamanan seperti melakukan fungsi *end-user authentication* dan mTLS.

e. Galley

Galley merupakan API penghubung dari sistem Istio dengan infrastruktur Kubernetes.

2.3.4 Kubernetes

Kubernetes merupakan platform yang bersifat *open source* yang digunakan untuk *orchestration container* atau orkestrasi kontainer secara *clustering* (Burns et al., 2019). Maksud dari orkestrasi adalah untuk melakukan manajemen sebuah kontainer secara otomatis mulai dari proses *computing*, *deployment*, penjadwalan atau *scheduling*, sampai dengan manajemen untuk bagian *networking*. Secara umum komponen arsitektur Kubernetes dibagi menjadi tiga yaitu:

a. Master Komponen

Bagian master menyediakan fungsi *control-plane* pada master Kubernetes.

1. Kube-apiserver

   Komponen yang bertugas untuk membuka API pada Kubernetes.

2. Etcd

   Bertugas untuk melakukan penyimpanan *key value* yang digunakan Kubernetes untuk menyimpan informasi pada sebuah klaster.

3. Kube-scheduler

   Mempunyai tugas untuk mengatur penjadwalan dan pembagian suatu kontainer pada Kubernetes pada tiap-tiap nodenya.

4. Kube-controller-manager

Salah satu komponen utama pada master Kubernetes yang bertugas sebagai *controller*. Pada bagian *controller* tersebut terdiri dari beberapa *controller* yaitu:

1. Node Controller

Memberikan informasi dan pengecekan pada setiap *node.*

2. Replication Controller

   Memberikan respon pengecekan pada setiap replika yang ada pada container

3. Endpoint Controller

   Memberikan informasi untuk objek *endpoint* pada setiap *Service*.

4. *Service* Account dan Token Controller

   Mengatur dan memberikan informasi untuk token hak akses pada API Kubernetes untuk setiap pembuatan *Namespace* baru.

5. Cloud-controller-manager

   Sama seperti Kube-controller-manager namun biasanya sudah mengalami kustomisasi dari pihak penyedia komputasi awan.

b. Node Komponen

Komponen-komponen utama pada setiap node Kubernetes.

1. Kubelet

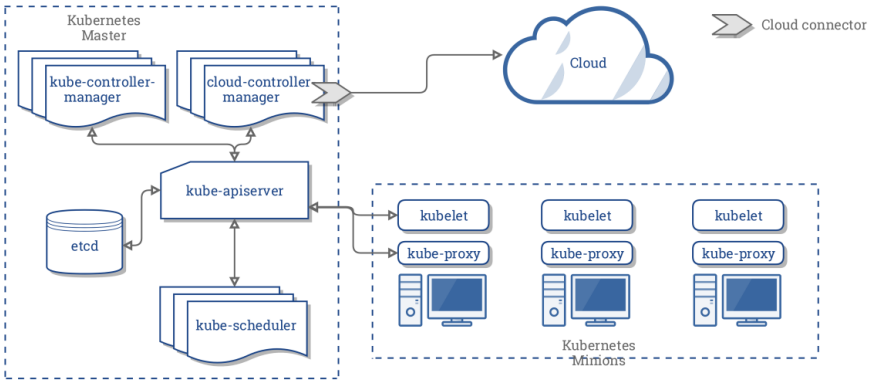
   Mempunyai fungsi sebagai pengecekan kontainer pada setiap *Pod*.

2. Kube-proxy

   Melakukan tugas untuk mengatur komunikasi dari sebuah kontainer atau *Pod* kepada *Service* dan melakukan *port forwarding.*

3. Container Runtime

   Merupakan aplikasi yang mempunyai fungsi untuk menjalankan kontainer. Beberapa kontainer runtime yang didukung baik oleh Kubernetes adalah Docker, containerd, cri-o, dan rktlet. Gambar 2.3 merupakan gambar untuk arsitektur Kubernetes.



Gambar 2.3 Arsitektur dan komponen pada Kubernetes

c. Addon

Merupakan tambahan fitur utama untuk mendukung komponen-komponen utama pada Kubernetes dan biasanya *addon* akan membuat *Namespace* sendiri dan langsung berkomunikasi dengan *Namespace* kube-system (The Kubernetes Authors, 2020).

Selain dari beberapa komponen arsitektur Kubernetes di atas, Kubernetes juga memiliki sejumlah komponen konfigurasi objek yang digunakan untuk melakukan konfigurasi dan administrasi. Di bawah ini merupakan beberapa objek tersebut.

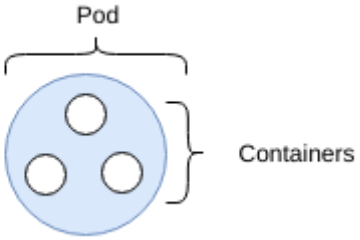
1. *Pods*

Sebuah *Pod* adalah unit dasar pada Kubernetes object yang dapat dibuat dan dilakukan *Deployment*. *Pod* membungkus sebuah kontainer (atau, di beberapa kasus, beberapa kontainer), sumber penyimpanan, alamat jaringan *IP* yang unik, dan opsi yang mengatur bagaimana kontainer harus dijalankan (Burns et al., 2019). *Pod* merupakan representasi dari unit *Deployment*: sebuah instance aplikasi di dalam Kubernetes, yang mungkin terdiri dari satu kontainer atau sekumpulan kontainer yang berbagi *resource*.

Pod di Kubernetes klaster dapat digunakan dengan dua cara, yaitu:

- *Pod* menjalankan satu kontainer
- *Pod* menjalankan beberapa kontainer yang perlu berjalan bersamaan

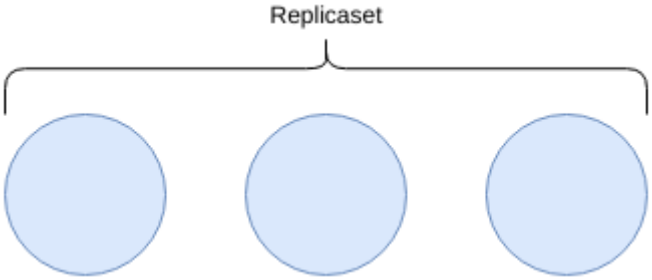
Gambar 2.4 merupakan visualisasi tentang Kubernetes *Pod*s.



Gambar 2.4 Kubernetes *Pod*s

2. *ReplicaSet*

ReplicaSet adalah objek pada Kubernetes yang digunakan untuk melakukan fungsi replikasi pada *Pod*s (Burns et al., 2019). Pada konfigurasi *ReplicaSet* dapat ditentukan sesuai dengan keinginan berapa banyak *Pod*s yang akan di replika. Ketika *ReplicaSet* bekerja, maka akan mengambil fungsi konfigurasi *Pod template*. Gambar 2.5 merupakan visualisasi tentang *ReplicaSet*.



Gambar 2.5 Kubernetes *ReplicaSet*s

3. *Service*

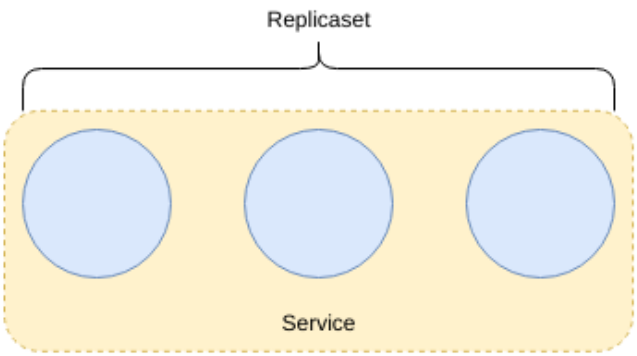
Service merupakan tingkat abstraksi objek dimana mendefinisikan kontainer atau *Pod*s untuk berkomunikasi dengan jaringan diluar maupun jaringan secara lokal di dalam klaster Kubernetes (Arundel & Domingus, 2019). Beberapa *Pod*s memiliki tujuan atau ditargetkan oleh *Service*. Target antara *Pod*s dan *Service* biasanya didefinisikan berdasarkan Label Selector.

Terdapat banyak metode atau cara untuk melakukan komunikasi eksternal (di luar klaster Kubernetes) yaitu dengan

- ClusterIP
- Node*Port*
- LoadBalancer

- ExternalName

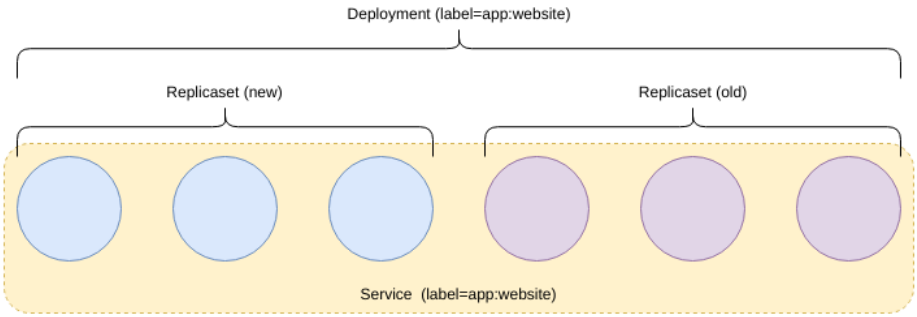
Gambar 2.6 merupakan visualisasi tentang *Service* pada Kubernetes.



Gambar 2.6 Kubernetes *Service*

4. *Deployment*

Deployment merupakan salah satu objek konfigurasi yang ada di Kubernetes untuk melakukan konfigurasi pembuatan *Pod*s dan mendukung pembaharuan *Pod*s dan *ReplicaSet*s. Selain mendukung dua pembuatan objek sekaligus, *Deployment* dapat mendukung untuk melakukan fungsi *rollback* pada *Pod*s (Arundel & Domingus, 2019). Gambar 2.7 merupakan visualisasi tentang *Deployment*.



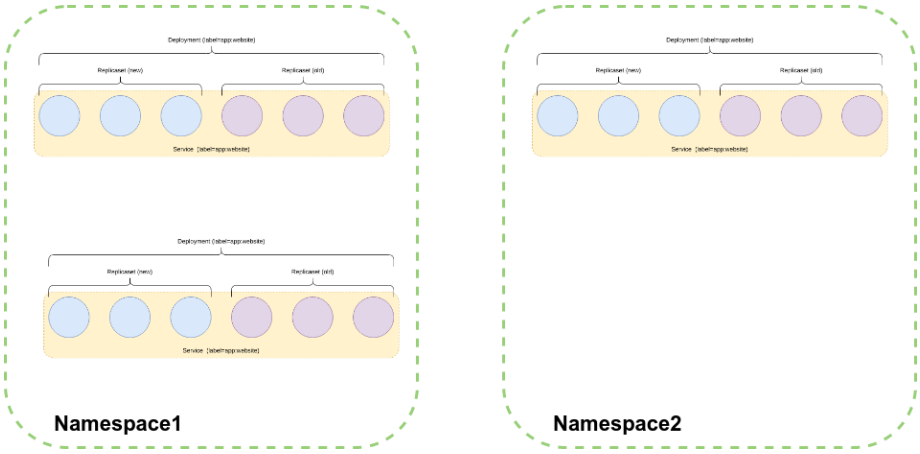
Gambar 2.7 Kubernetes *Deployment*

5. *Namespace*

Namespace merupakan salah satu objek yang ada di dalam klaster Kubernetes untuk melakukan pengelompokan dari berbagai macam objek yang ada di Kubernetes (Arundel &

Domingus, 2019). Dengan adanya *Namespace* ini, dapat membantu untuk melakukan manajemen *environment* dengan banyak pengguna yang akan memakai klaster Kubernetes.

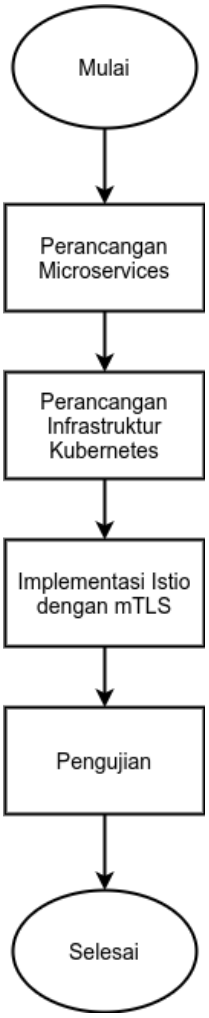
Selain untuk manajemen environment, *Namespace* juga dapat digunakan untuk tujuan pengaturan resource penggunaan objek yang ada di Kubernetes sehingga tidak mengganggu *resource* objek yang lain didalam *Namespace* yang berbeda. Gambar 2.8 merupakan visualisasi tentang *Namespace*.



Gambar 2.8 Kubernetes *Namespace*

BAB III
METODOLOGI PENELITIAN

3.1 Metode Penelitian



Gambar 3.1 Metode Penelitian

3.2 Perancangan aplikasi dan infrastruktur

Berdasarkan pada Gambar 3.1 Metode Penelitian, penjelasan untuk tiap langkah pada metode penelitian akan dijabarkan pada subbab-subbab berikut.

3.2.1 Perancangan *Microservices*

Pada subbab ini akan membahas perancangan tentang desain rancangan *microservices* yang digunakan sebagai bukti konsep atau *proof of concept* dalam implementasi *mutual*

Transport Layer Security (mTLS). Dalam sub bab perancangan *microservices* sendiri akan dibagi menjadi dua tahapan yaitu:

1. Rancangan *Services*

Rancangan *microservices* yang akan diimplementasikan dalam penelitian ini adalah aplikasi untuk pengecekan cuaca yang ada di Kota Malang lalu akan menampilkan status kondisi cuaca dan suhu yang ada di kota tersebut. Pada proses perancangan *microservices* ini terdapat dua *services* yang akan berjalan, yaitu:

a. *Frontend*

Pada *microservices frontend* akan berisi tentang tampilan sederhana dengan menampilkan informasi tentang nama kota, kondisi cuaca, dan suhu. *Service frontend* sendiri akan ditulis dengan framework Nuxt.JS. Kemudian *Service frontend* ini akan mengonsumsi API internal dari *service backend*. *Service frontend* sendiri akan membutuhkan *package* atau *library*, yaitu Axios untuk melakukan *request*.

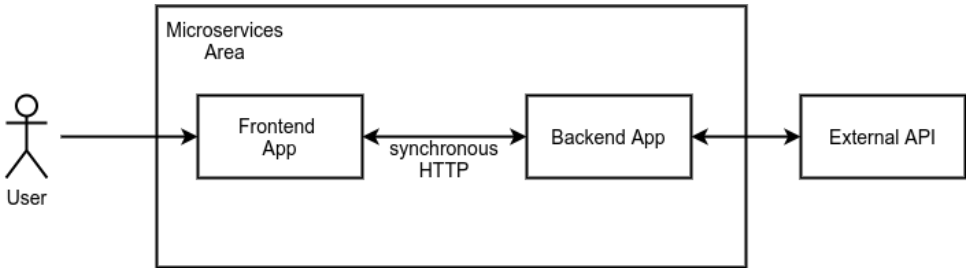
b. *Backend*

Untuk *microservices backend* ini akan dibuat dengan *framework* Flask (Python) dengan tambahan *module request* dan flask_cors. *Service backend* akan melakukan *request* ke penyedia API cuaca Openweathermap sebagai API eksternal untuk mendapatkan data atau informasi cuaca dengan format JSON.

2. *Building container*

Tahapan *building container* ini adalah proses merubah dari dari aplikasi tradisional ke dalam kontainer. Dari rancangan di atas terdapat dua *services* yang akan berjalan di dalam rancangan *microservices*, maka akan ada dua buah aplikasi atau *services* dalam bentuk kontainer. Sebuah kontainer akan membutuhkan *repository* atau tempat untuk menyimpan *image* kontainer tersebut. Terdapat banyak *repository* untuk menyimpan *image* kontainer baik yang bersifat *private* maupun *public*, namun untuk penelitian tentang implementasi *mutual Transport Layer Security* (mTLS) ini akan menggunakan Docker Hub.

Dari dua tahapan pada subbab perancangan *microservices* diatas, kedua *services* atau aplikasi akan saling berkomunikasi secara *synchronous* yaitu teknik berkomunikasi antar *services* secara langsung dengan protokol HTTP. Gambar 3.2 merupakan bentuk rancangan dua aplikasi dengan arsitektur *microservices*.



Gambar 3.2 Rancangan *Microservices*

3.2.2 Perancangan Infrastruktur Kubernetes

Setelah perancangan *microservices*, tahapan selanjutnya adalah menentukan perancangan infrastruktur Kubernetes. Aplikasi atau *services* yang sudah di *pack* ke dalam kontainer akan berjalan di atas infrastruktur Kubernetes yang memiliki fungsi untuk melakukan kegiatan orkestrasi kontainer. Terdapat banyak cara atau metode untuk menjalankan Kubernetes, seperti dalam bentuk *virtual machine*, *bare metal* maupun menggunakan *cloud provider*. Untuk perancangan konfigurasi infrastruktur Kubernetes akan menggunakan dua objek konfigurasi yang ditulis dengan format YAML yaitu:

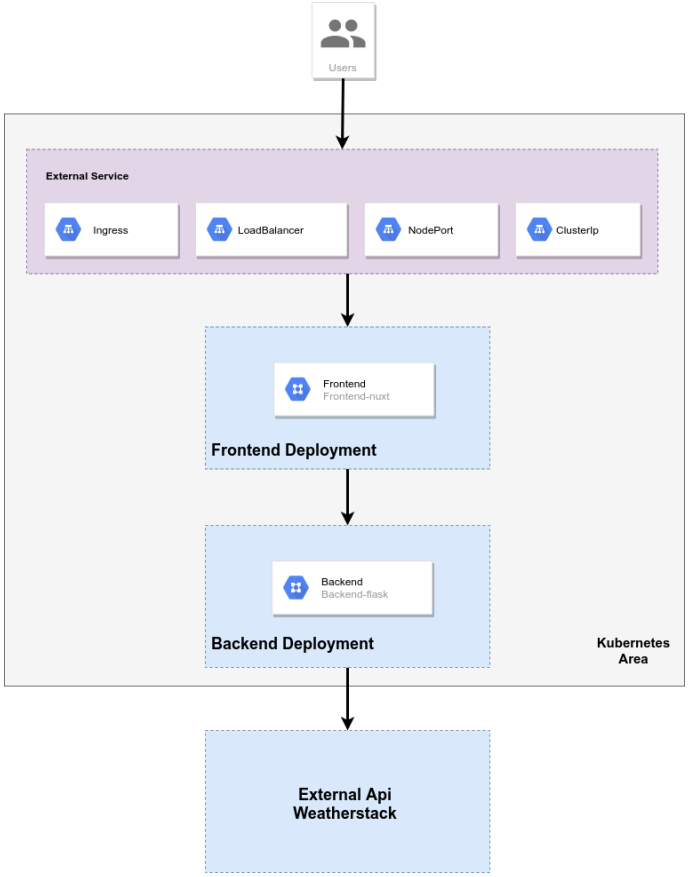
1. *Deployment*

Deployment merupakan salah satu jenis objek konfigurasi pada Kubernetes yang berfokus untuk melakukan pembuatan sebuah pods, sedangkan pods itu sendiri adalah sekumpulan kontainer yang berjalan di atas Kubernetes. Dari rancangan *microservices* di atas, maka akan ada dua file konfigurasi *deployment*.

2. *Service*

Service adalah salah satu jenis objek konfigurasi pada Kubernetes yang memiliki fungsi untuk membuat jembatan komunikasi antar pods maupun dengan eksternal *service* baik antar *service* yang ada di dalam Kubernetes maupun di luar klaster Kubernetes. Dari rancangan *microservices* di atas, maka akan ada dua objek konfigurasi *service*.

Dari objek konfigurasi diatas, merupakan salah satu bentuk mengintegrasikan kontainer yang sudah dirancang ke dalam klaster Kubernetes nantinya. Gambar 3.4 merupakan bentuk rancangan infrastruktur *microservices* yang akan berjalan di atas klaster Kubernetes.



Gambar 3.3 Rancangan Infrastruktur Kubernetes

3.3 Implementasi Istio dengan mutual Trans*port* Layer Security (mTLS)

Istio yang digunakan untuk implementasi *mutual Transport Layer Security* (mTLS) ini akan berjalan diatas infrastruktur Kubernetes. Menurut halaman dokumentasi resmi, Istio dapat memberikan akses dan aturan di atas Kubernetes dengan memanfaatkan fitur CRD yang ada di Kubernetes klaster. CRD (*Custom Resources Definition*) adalah kumpulan API yang dapat bersifat sebagai *extensions* pada Kubernetes API sehingga *user* atau admin infrastruktur Kubernetes dapat melakukan kustomisasi pada Kubernetes API agar dapat digunakan secara terus menerus atau *continuous*.

Dalam proses implementasi ini akan menggunakan beberapa file konfigurasi *manifest* yang ada pada Istio dengan format penulisan YAML. Di bawah ini merupakan beberapa file konfigurasi *manifest* yang akan digunakan.

1. Gateway

Service mesh mempunyai salah satu ciri, yaitu kemampuan untuk mengisolasi *microservices* yang ada di dalamnya, sehingga memerlukan sebuah pintu untuk memberikan akses untuk *traffic* masuk ketika seorang pengguna akan mengakses *services* yang ada di

dalam *service mesh*. Fungsi *gateway* di sini adalah sebagai *load balancer* dan mendeskripsikan *port* apa saja yang akan dilewati dari *traffic* yang masuk.

2. *Virtual Service*

Setelah keluar dari *gateway load balancer*, selanjutnya adalah bagaimana cara meneruskan *traffic* yang sudah masuk melalui *gateway* ke dalam aplikasi *microservices* pada *service mesh*. *Virtual service* memiliki fungsi untuk melakukan *routing* dari *gateway* ke *services* pertama yang ada di dalam *service mesh*. Pada bagian perancangan awal, aplikasi *microservices frontend* adalah sebagai *service* atau aplikasi pertama yang menerima trafik masuk dari *gateway*.

3. *Destination Rule*

Destination rule merupakan salah satu objek konfigurasi *manifest* yang ada di Istio dengan fungsi untuk menerapkan aturan atau *policy* ke dalam *services* yang sudah diatur *traffic routing*-nya dan juga dapat melakukan pemilihan algoritma *load balancer* ketika *microservices* yang ada di *service mesh* mengalami *scaling*. *Destination rule* ini nantinya akan mengatur *policy routing* ketika aplikasi atau *services* berkomunikasi dengan *mutual Transport Layer Security* (mTLS).

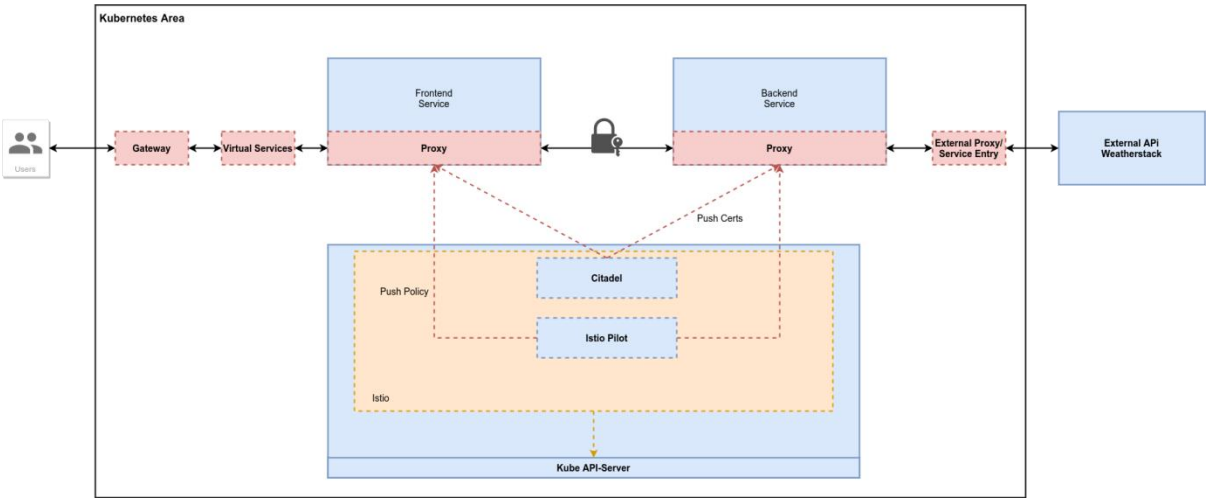
4. *Policy*

Policy pada objek konfigurasi *manifest* Istio mempunyai fungsi untuk mendefinisikan aturan keamanan tentang bagaimana proses autentikasi setiap *microservices* yang ada di *service mesh* bekerja. Pendefinisian *policy* ini akan diterapkan nantinya pada *services* backend sebagai target dari autentikasinya dan memberikan aturan tambahan tentang protokol yang harus digunakan ketika aplikasi *frontend* berkomunikasi dengan *backend*.

5. *Service Entry*

Dari rancangan aplikasi *microservices* di atas, *services backend* akan berkomunikasi dengan eksternal API milik Openweathermap untuk mendapatkan data cuaca. *Service entry* memiliki fungsi untuk memberikan aturan konfigurasi tentang *services* eksternal apa saja yang dapat berkomunikasi dengan *service mesh* yang memiliki sifat *isolate*. Dalam hal ini, *services* yang bersifat sebagai *traffic* menuju keluar atau *egress*.

Dari beberapa komponen objek konfigurasi yang dimiliki Istio, aplikasi atau *services* akan diintegrasikan dari infrastruktur Kubernetes ke dalam perancangan *service mesh* menggunakan Istio. Gambar 3.4 merupakan bentuk perancangan *service mesh* dengan Istio dengan implementasi *mutual Transport Layer Security* (mTLS).



Gambar 3.4 Implementasi mutual Trans*port* Layer Security (mTLS)

3.4 Pengujian

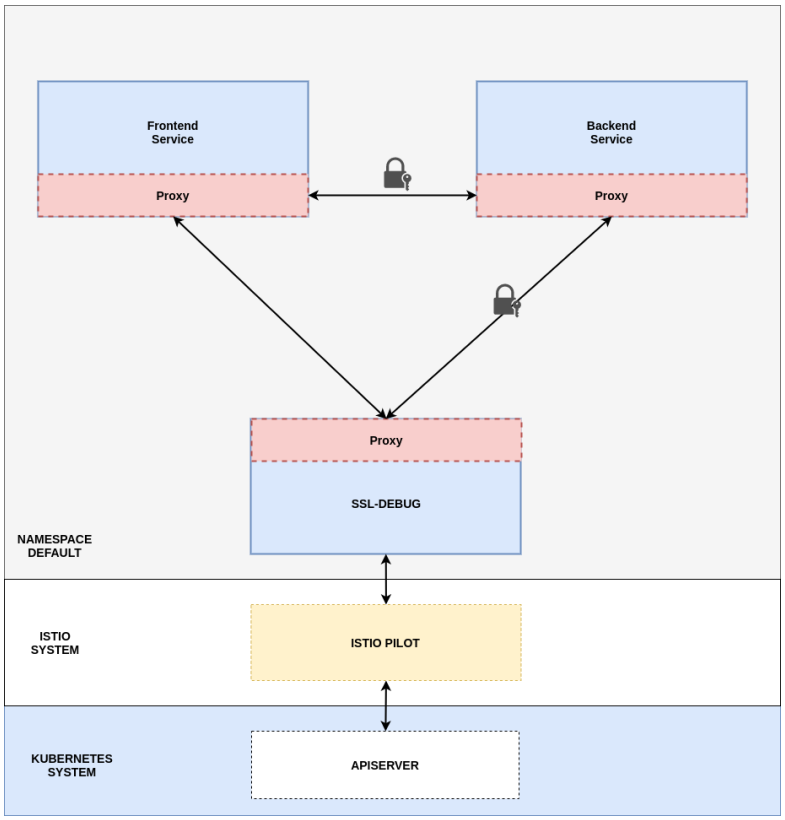
Pada tahapan pengujian ini akan dilakukan dengan dua metode. Metode pertama adalah verifikasi *mutual Transport Layer Security* (mTLS) dimana metode ini adalah untuk melakukan proses pemeriksaan ketika komunikasi di dalam *microservices* sudah dalam kondisi terenkripsi dengan mTLS atau tidak. Adapun Tabel 3.1 menunjukkan beberapa *tools* yang akan digunakan untuk pengujian tersebut.

Tabel 3.1 *Tools* yang digunakan untuk pengujian

Tools	**Jenis**	**Deskripsi**	**Tautan/Sumber**
ksniff	*Open source*	Merupakan salah satu plugin tambahan yang mempunyai fungsi untuk melakukan *capture* aktifitas *network* pada pod di Kubernetes	https://github.com/eldadru/ksniff
Wireshark	*Open source*	*Tool* yang digunakan untuk menganalisa *packet* pada suatu jaringan	https://www.wireshark.org/
Kiali	*Open source*	Tool yang digunakan untuk visualisasi *microservices* dan *service mesh*	https://github.com/kiali/kiali
Ssldump	*Open source*	*Tool* yang digunakan untuk menganalisa protokol SSL/TLS pada suatu jaringan	https://linux.die.net/man/1/ssldump
Testssl.sh	*Open source*	Salah satu *tool* yang mempunyai fungsi untuk melakukan pengecekan	https://testssl.sh/

		pada protokol dan ciphers SSL/TLS	
Mercury	*Open source*	Tool yang dapat digunakan untuk melakukan proses fingerprinting dan analisis paket pada jaringan	https://github.com/cisco/mercury

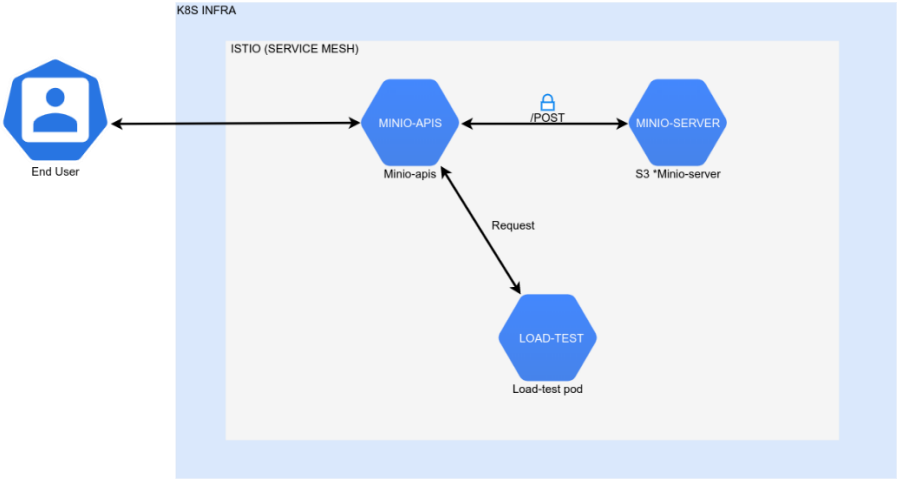
Dari beberapa tools yang akan digunakan, penggunaan tool testssl.sh akan dilakukan dengan mengubah ke dalam bentuk kontainer dan akan menjadi objek konfigurasi pod pada klaster infrastruktur Kubernetes dengan tujuan agar bisa berkomunikasi dengan komponen objek API dari Kubernetes maupun komponen objek konfigurasi Istio dan aplikasi atau *service*s yang berjalan di atas Kubernetes harus dalam bentuk kontainer. Gambar 3.5 merupakan bentuk rancangan arsitektur tool yang sudah dalam bentuk objek konfigurasi pod di atas klaster Kubernetes



Gambar 3.5 Rancangan pod untuk tool pengujian

Kemudian untuk pengujian selanjutnya akan mengukur bagaimana performa dari *microservices* yang sudah menggunakan *mutual Transport Layer Security* (mTLS) dengan

yang tidak menggunakan mTLS. Rancangan pengujian performa ini akan mencoba melakukan 100 *request* terhadap arsitektur *microservices* yang sudah dibuat dengan cara *request method* POST dari API ke S3 *server* dengan proses mengunggah berkas secara acak dan untuk rancangan diagram pengujian performa seperti pada Gambar 3.6.



Gambar 3.6 Diagram pengujian Apache Jmeter

Seperti pada Gambar 3.6, pengujian kedua ini akan menggunakan *tool* Apache Jmeter dengan memperhatikan beberapa parameter pengujian, seperti *Response Time*, *Throughput*, dan *Error Rate*.

BAB IV
HASIL DAN PEMBAHASAN

4.1 Implementasi Micro*services*

Tahapan pertama adalah implementasi *microservices* yang akan digunakan untuk *proof of concept* dari proses pengerjaan implementasi *mutual Transport Layer Security* (mTLS). Dalam tahapan implementasi *microservices*, penulis memaparkan penjelasan untuk setiap tahap dalam subbab-subbab berikut.

4.1.1 Frontend

Pada tahapan implementasi *microservices frontend* ini, penulis menuliskan ke dalam kode bahasa javascript yang menggunakan *framework* Nuxt.Js. *Service frontend* ini akan mengkonsumsi API dari *service backend*. Gambar 4.1 merupakan potongan kode yang ada di file nuxt.config.js ketika *service frontend* berkomunikasi dengan *service backend*.

```
export default {
  head: {
    link: [
      { rel: 'icon', type: 'image/x-icon', href: '/favicon.ico' },
      {rel:       'stylesheet',       type:       'text/css',       href:
'https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0/css/bootstrap.min.css'}
    ]
  },
  env: {
    baseUrl: process.env.BASE_URL || 'http://backend-svc:5000'
  },
```

Gambar 4.1 Kode program nuxt.config.js

Pada bagian baseUrl mengarahkan ke http://backend-svc:5000 dimana backend-svc merupakan alamat *service backend* yang nantinya akan diterapkan di dalam Kubernetes. Kemudian didalam *service frontend* ini menggunakan *package* tambahan untuk melakukan *request* API json, yaitu dengan *package* axios. Gambar 4.2 merupakan potongan kode program untuk axios.

```
<script>
import axios from 'axios'
export default {
  async asyncData () {
     const { data } = await axios.get(process.env.baseUrl +'/api/cuaca')
     return { weather: data }
   },
 }
</script>
```

Gambar 4.2 Potongan kode axios

Gambar 4.2 menunjukkan bahwa axios akan berkomunikasi dengan API *backend* dengan *endpoint* /api/cuaca dengan cara melakukan request dengan tipe metode GET dan mendapatkan format keluaran JSON yang disimpan di dalam variabel *weather*.

4.1.2 *Backend*

Tahapan membuat *service backend* yang ditulis menggunakan bahasa python dengan *framework* Flask. *Service backend* ini akan melakukan *request* ke API eksternal dengan API key yang diperoleh ketika mendaftar dari *website* Openweathermap. Gambar 4.3 merupakan potongan kode dari *service backend*.

```
@app.route('/api/cuaca', methods=['GET'])
def cuaca():
    API_KEY = '<Key api owm>'
    CITY = 'Malang'
    r                                                                    =
"https://api.openweathermap.org/data/2.5/weather?q={}&appid={}&units=metric
".format(CITY, API_KEY)
    api_response = requests.get(r).json()
    return jsonify(api_response)
```

Gambar 4.3 Potongan kode *service backend*

Dari rancangan *query* akan mengambil data cuaca dari kota Malang lalu data yang berupa JSON akan diteruskan dan disimpan ke dalam variabel api_response. *Service* backend ini akan mempunya *endpoint* /api/cuaca yang kemudian bisa diakses oleh *service* frontend.

4.1.3 *Building Container*

Tahapan *building container* ini membahas proses penulisan Dockerfile, proses *build* kontainer sampai dengan proses *push* kontainer ke dalam kontainer *registry*. Dari uraian pembahasan subbab implementasi *microservices* di atas, terdapat dua file Dockerfile, yaitu

untuk Dockerfile *frontend* dan Dockerfile *backend*. Gambar 4.4 merupakan Dockerfile *service frontend*.

```
FROM node:12.14.1-alpine3.11
RUN mkdir -p /usr/src/nuxt-app
WORKDIR /usr/src/nuxt-app

RUN apk update && apk upgrade
RUN apk add git
COPY . /usr/src/nuxt-app/
RUN npm install
RUN npm install axios
RUN npm run build

EXPOSE 3000
ENV NUXT_HOST=0.0.0.0
ENV NUXT_PORT=3000
CMD [ "npm", "start" ]
```

Gambar 4.4 Dockerfile *service frontend*

Dari Dockerfile *service frontend*, kontainer *service frontend* nantinya akan berjalan pada *port* 3000. Selanjutnya adalah Dockerfile untuk *service backend*, Gambar 4.5 merupakan Dockerfile dari *service backend*.

```
FROM python:3.5 AS build
COPY requirements.txt .
RUN pip install -r ./requirements.txt

FROM gcr.io/distroless/python3
COPY --from=build /usr/local/lib/python3.5/site-packages/ \
 /usr/lib/python3.5/.

WORKDIR /usr/src/app
COPY app.py app.py
CMD ["-m", "flask", "run", "--host", "0.0.0.0"]
```

Gambar 4.5 Dockerfile *service backend*

Uraian untuk Gambar 4.5 adalah *service backend* menggunakan fitur *multi-stage build*s, yaitu terdapat penggunaan variabel FROM lebih dari satu kali untuk mendukung file kontainer yang lebih kecil. Untuk *service backend* sendiri kontainernya nantinya akan berjalan di *port default* dari *framework* flask yaitu *port* 5000. Setelah persiapan Dockerfile, langkah berikutnya adalah proses *build* kontainer *service frontend*. Gambar 4.6 adalah perintah untuk *build* kontainer *service frontend*.

```
~$ docker build -t riskiwah/frontend-forecast:owm-k8s .
```

Gambar 4.6 *Build* kontainer *service frontend*

Kontainer *service frontend* tersebut memiliki nama sebagai *frontend-forecast* dengan *tag* owm-k8s. Kemudian adalah *build* kontainer untuk *service backend*, pada Gambar 4.7merupakan perintah untuk *build* kontainer *service backend* dan akan diberi nama *backend-forecast* dengan *tag* owm.

```
~$ docker build -t riskiwah/backend-forecast:owm .
```

Gambar 4.7 *Build* kontainer *service backend*

Tahapan setelah melakukan *build* kontainer adalah melakukan *push* atau unggah *image* kontainer yang sudah di-*build* ke dalam *container registry* yang nantinya akan *image* kontainer tersebut bisa diterapkan ke dalam infrastruktur Kubernetes. Gambar 4.8 merupakan perintah untuk melakukan unggah ke dalam *container registry*.

```
docker push riskiwah/frontend-forecast:owm-k8s
```

Gambar 4.8 *Push* kontainer *service frontend*

```
docker push riskiwah/backend-forecast:owm
```

Gambar 4.9 *Push* kontainer *service backend*

4.2 Implementasi Infrastruktur Kubernetes

Tahap selanjutnya adalah implementasi infrastruktur Kubernetes. Penjelasan untuk setiap tahapan akan dijelaskan dalam subbab-subbab di bawah ini.

4.2.1 Membuat Instance Google Kubernetes Engine

Tahap ini adalah membuat instance atau *virtual machine* yang ada di dalam Google Cloud Platform (GCP) dengan produknya untuk Kubernetes adalah Google Kubernetes Engine (GKE). Gambar 4.10 merupakan perintah untuk membuat instance Kubernetes pada Google Kubernetes Engine (GKE).

```
~$ gcloud container --project "istio-lab-267501" \
clusters create "istio-lab-testing" --zone "asia-southeast2-a" \
--no-enable-basic-auth --cluster-version "1.16.9-gke.6" \
--machine-type "e2-medium" --image-type "UBUNTU" \
--disk-type "pd-standard" --disk-size "100" --num-nodes "4" \
--enable-stackdriver-kubernetes --enable-ip-alias \
--network "projects/istio-lab-267501/global/networks/default" \
--subnetwork                    "projects/istio-lab-267501/regions/asia-
southeast2/subnetworks/default" \
--default-max-pods-per-node "110" \
--enable-autoupgrade --enable-autorepair
```

Gambar 4.10 Membuat instance Google Kubernetes Engine

Penjelasan setiap perintah membuat instance Google Kubernetes Engine, dapat dilihat pada Tabel 4.1.

Tabel 4.1 Penjelasan perintah GKE

No	Perintah / Command	Keterangan
1	gcloud container	Masuk ke dalam sub perintah dari gcloud
2	--project "istio-lab-267501"	Klaster yang akan dibuat berada dalam project istio-lab-267501
3	clusters create "istio-lab-testing"	Membuat klaster GKE dengan nama istio-progress
4	--zone "asia-southeast2-a"	Tempat dari klaster berada di asia-southeast2-a
5	--no-enable-basic-auth	Tidak mengaktifkan autentikasi admin Kubernetes
6	--cluster-version "1.16.9-gke.6"	Versi klaster yang dipakai
7	--machine-type "e2-medium"	Spesifikasi instance atau VM
8	--image-type "UBUNTU"	Base sistem operasi yang digunakan setiap instance
9	--disk-type "pd-standard"	Jenis penyimpanan yang digunakan setiap instance
10	--disk-size "100"	Ukuran penyimpanan yang dipakai setiap instance
11	--num-nodes "4"	Jumlah node yang dibuat

4.2.2 Menerapkan Kubernetes Manifest

Tahap ini akan menjelaskan tentang menerapkan Kubernetes *manifest* dari *service frontend* dan *backend*. Gambar 4.11 merupakan implementasi potongan format YAML untuk objek konfigurasi deployment Kubernetes *service frontend*.

```
...
kind: Deployment
metadata:
  name: frontend-dep
...
    spec:
      containers:
      - name: nuxt
        image: riskiwah/frontend-forecast:owm-k8s
        ports:
          - containerPort: 3000
```

Gambar 4.11 Implementasi deployment *frontend*

Dari kode program pada Gambar 4.11, objek konfigurasi deployment *frontend* akan menggunakan *image* kontainer yang sudah dibuat dan diunggah dari *container registry* Docker Hub. Deployment *frontend* ini akan berjalan dengan *port* 3000. Selanjutnya, Gambar 4.12 adalah potongan format YAML untuk objek konfigurasi *deployment* Kubernetes *service backend*.

```
...
kind: Deployment
metadata:
  name: backend-dep
...
    spec:
      containers:
      - name: flask
        image: riskiwah/backend-forecast:owm
        ports:
        - containerPort: 5000
```

Gambar 4.12 Implementasi deployment *backend*

Objek konfigurasi deployment *backend* pada Gambar 4.12 akan mengambil atau mengunduh *image* kontainer *service backend* yang sudah di-*push* ke dalam *container registry*. Setelah dua objek *deployment* sudah dibuat, pod yang ada pada objek konfigurasi deployment tersebut harus bisa diakses dari luar. Gambar 4.13 merupakan potongan format YAML dari objek konfigurasi *service frontend* agar bisa diakses di luar pod.

```
kind: Service
...
spec:
..
  type: ClusterIP
  ports:
  - name: http-nuxt
    port: 3000
    targetPort: 3000
```

Gambar 4.13 Implementasi *service frontend*

Dari Gambar 4.13, jenis objek konfigurasi *service* yang digunakan adalah ClusterIP yang merupakan salah satu bentuk tipe dari *service* yang secara *default* diberikan oleh Kubernetes, tipe ini akan hanya memberikan akses eksternal hanya untuk *range* alamat IP yang digunakan oleh klaster infrastruktur Kubernetes. Dalam objek konfigurasi *service frontend* ini akan tetap menggunakan *port* 3000 ketika diakses secara eksternal. Gambar 4.14 adalah potongan format YAML untuk objek *service* aplikasi *backend*.

```
kind: Service
...
  name: backend-svc
spec:
...
  type: ClusterIP
  ports:
  - name: http-flask
    port: 5000
    targetPort: 5000
```

Gambar 4.14 Implementasi *Service backend*

Dari dua objek *manifest* deployment dan *service* pada Gambar 4.14, perlu diterapkan ke dalam klaster Kubernetes dengan perintah *tool* kubectl yang dapat dilihat pada Gambar 4.15.

```
~$ kubectl apply -f <nama file>.yaml
```

Gambar 4.15 Menerapkan Kubernetes *manifest*

4.3 Implementasi Istio dengan *mutual Transport Layer Security* (mTLS)

Tahap ini merupakan tahap mengimplementasikan *mutual Transport Layer Security* (mTLS) pada infrastruktur Kubernetes yang sudah dibuat. Pada tahap ini akan membahas tentang proses pemasangan Istio pada infrastruktur Kubernetes, injeksi *namespace* Kubernetes dan penerapanya yang akan dijelaskan dalam subbab-subbab berikut.

4.3.1 Memasang Istio

Tahap memasang Istio adalah tahap awal yaitu langkah pertama membuat *namespaces* sebagai tempat pod dan semua konfigurasi *manifest* Istio berada. Perintah untuk membuat *namespaces* dapat dilihat pada Gambar 4.16.

```
~$ kubectl create namespace istio-system
```

Gambar 4.16 Membuat *namespaces*

Tahapan kedua adalah melakukan instalasi atau pemasangan *Custom Resource Definitions* (CRD) agar nantinya kustomisasi API dari Istio dapat berjalan di dalam infrastruktur Kubernetes. Gambar 4.17 adalah perintah memasang CRD Istio dengan menggunakan *tool* helm.

```
~$ helm template install/kubernetes/helm/istio-init --name istio-init --
namespace istio-system | kubectl apply -f -
```

Gambar 4.17 Memasang *Custom Resource Definitions* (CRD)

Setelah proses pemasangan *Custom Resource Definitions* (CRD), tahap selanjutnya adalah instalasi semua komponen objek *manifest* yang dimiliki Istio. Dalam proses ini, penulis memilih *profile* instalasi istio-demo karena dalam *profile* demo semua fitur utama Istio akan diaktifkan. Gambar 4.18 merupakan perintah untuk instalasi objek dan pemilihan *profile* instalasi dari Istio.

```
~$ helm template install/kubernetes/helm/istio --name istio --namespace
istio-system \
--values install/kubernetes/helm/istio/values-istio-demo.yaml | kubectl
apply -f -
```

Gambar 4.18 Memasang objek Istio

Tahap setelah pemasangan objek konfigurasi Istio adalah melakukan proses *inject* terhadap *namespaces* yang digunakan sebagai *services mesh*. Proses injeksi ini akan membuat tambahan pod pada pod yang sudah penulis *deploy* ke dalam Kubernetes, pod tersebut adalah istio-proxy yang akan digunakan sebagai jembatan komunikasi antar *service*s dan bersifat *sidecar*. Penulis menggunakan namespaces *default* pada infrastruktur Kubernetes sebagai

tempat *services mesh microservices frontend* dan *backend*. Gambar 4.19 adalah perintah untuk melakukan injeksi namespaces *default* dengan cara memberi label pada *namespaces*..

```
~$ kubectl label namespace default istio-injection=enabled
```

Gambar 4.19 Injeksi *namespaces*

4.3.2 Menerapkan Istio *Manifest*

Tahap ini merupakan penerapan objek *manifest* yang dimiliki Istio yang dapat berkomunikasi dengan Kubernetes API melalui *Custom Resource Definitions* (CRD). Tahapan pertama adalah dengan membuat objek konfigurasi *gateway* agar dapat diakses dari user luar atau secara *ingress*. Gambar 4.20 merupakan potongan format YAML dari objek Istio gateway.

```
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
...
  selector:
    istio: ingressgateway
  servers:
  - port:
      number: 80
      name: http
      protocol: HTTP
    hosts:
    - "default/*"
```

Gambar 4.20 Implementasi gateway

Secara *default*, *selector* Istio adalah *ingress-gateway* di mana nantinya *traffic* yang masuk (*ingress*) akan dilewatkan melalui objek pod dan *service ingress-gateway* miliki Istio yang berada di *namespaces* istio-system. Untuk *port* nantinya menggunakan *port default* 80 sebagai akses ke *service* mesh aplikasi *frontend*. Kemudian untuk membuat *service frontend* sebagai *service*s pertama yang dipakai secara *interface*, maka dibutuhkan objek konfigurasi Istio untuk melakukan *redirect traffic* dari objek *gateway* dan fungsi tersebut dimiliki oleh virtual *service*. Gambar 4.21 merupakan potongan format YAML untuk *virtual service*.

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
...
  gateways:
  - gw-app
  http:
  - match:
    - uri:
        exact: /
    route:
    - destination:
        host: frontend-svc
        port:
          number: 3000
```

Gambar 4.21 Implementasi *virtual service*

Dari potongan format YAML pada Gambar 4.21, tab *route* akan mengarahkan ke dalam objek *service* yang dimiliki oleh aplikasi *frontend* dengan *port* 3000 yang sudah didefinisikan sebelumnya. Kemudian untuk menerapkan komunikasi *mutual Transport Layer Security* (mTLS) adalah dengan menentukan *server* yang digunakan untuk autentikasi dua arah. *Service backend* digunakan sebagai *server side* dalam penerapan *mutual Transport Layer Security* (mTLS) ini. Objek konfigurasi *destination rule* dapat digunakan untuk menentukan *server side* untuk autentikasi dua arah yang dapat dilihat pada potongan format YAML seperti pada Gambar 4.22.

```
apiVersion: "networking.istio.io/v1alpha3"
kind: "DestinationRule"
...
spec:
  host: "backend-svc.default.svc.cluster.local"
  trafficPolicy:
    tls:
      mode: ISTIO_MUTUAL
```

Gambar 4.22 Implementasi *destination rule*

Objek konfigurasi *destination rule* pada Gambar 4.22, merujuk pada *host* backend-svc.default.svc.cluster.local yang dipakai oleh objek konfigurasi *service backend* dengan format penamaan DNS internal yang dipakai di dalam Kubernetes. Tab selanjutnya adalah menentukan *traffic policy* dengan mode aturan ISTIO_MUTUAL yang mempunyai arti semua *traffic* yang masuk ke dalam *service backend* harus berkomunikasi secara *mutual Transport Layer Security* (mTLS). Untuk menentukan *client side*, penulis menggunakan objek konfigurasi *policy* yang ada pada Gambar 4.23.

```
apiVersion: "authentication.istio.io/v1alpha1"
kind: "Policy"
...
  targets:
  - name: backend-svc
  peers:
  - mtls: {}
```

Gambar 4.23 Implementasi *policy*

Penjelasan dari objek konfigurasi pada Gambar 4.23 adalah ketika *client* atau dalam hal ini adalah *service frontend* ingin berkomunikasi dengan *service backend* harus memiliki izin berupa pertukaran sertifikat digital (*handshake*) dari sisi *server* dan *client* melalui verifikasi dari komponen objek Istio citadel dan Istio pilot, karena dua objek tersebut bertugas sebagai PKI (*Public Key Infrastructure*). Istio citadel berfungsi untuk melakukan *push* sertifikat digital ke masing-masing *services* yang ada di dalam *service mesh*, sedangkan Istio pilot mempunyai fungsi untuk melakukan verifikasi dan menentukan aturan komunikasi yang berjalan pada *service mesh*.

Dari dua komponen objek Istio yang sudah dibahas di atas, dua komponen objek tersebut akan meneruskan sebelumnya ke dalam Kubernetes API dan memanggil fungsi objek konfigurasi *service account* untuk melakukan *generate* komponen sertifikat digital dengan format x.509 Certificate dengan ditambahnya komponen *Subject Alternative Name* dengan format SPIFFE. SPIFFE merupakan *framework* keamanan yang digunakan untuk memberikan keamanan identitas yang ditambahkan pada komponen x.509 Certificate.

Selain memanggil fungsi objek konfigurasi *service account*, objek Istio citadel dan pilot juga memanggil objek konfigurasi *config map* yang mempunyai fungsi atau tujuan menyimpan hasil *generate* sertifikat digital yang nantinya di-*push* ke dalam kontainer *sidecar* istio-proxy melalui komponen objek Istio pilot.

Tahapan terakhir adalah menentukan *traffic* komunikasi ke arah luar atau *egress* karena *service backend* membutuhkan komunikasi API secara eksternal ke penyedia API cuaca Openweathermap. Gambar 4.24 merupakan potongan format YAML objek konfigurasi *service* entry untuk *traffic* yang mengarah ke luar.

```
apiVersion: networking.istio.io/v1alpha3
kind: ServiceEntry
...
  - api.openweathermap.org
  ports:
  - number: 80
    name: http
    protocol: HTTP
  - number: 443
    name: https
    protocol: HTTPS
  location: MESH_EXTERNAL
```

Gambar 4.24 Implementasi *service* entry

Dari uraian pembahasan beberapa objek konfigurasi yang dimiliki oleh Istio untuk implementasi *mutual Transport Layer Security* (mTLS) harus diterapkan ke dalam infrastruktur klaster Kubernetes dengan perintah seperti pada Gambar 4.25.

```
~$ kubectl apply -f <nama file>.yaml
```

Gambar 4.25 Menerapkan Istio *manifest*

4.4 Pengujian

Tahap terakhir adalah tahap pengujian yang akan dilakukan sesuai metode pengujian yang ada di bab sebelumnya. Sebelum melakukan tahap pengujian, penulis membuat *tool* dalam bentuk kontainer agar mempermudah proses pengujian karena semua yang berjalan diatas klaster Kubernetes harus dalam bentuk kontainer. Gambar 4.26 merupakan Dockerfile yang dipakai sebagai *tool* melakukan pengecekan pada SSL/TLS.

```
FROM alpine:3.11
RUN apk update && apk upgrade && apk add --no-cache \
    bash \
    procps \
    drill \
    git \
    coreutils \
    libidn \
    curl \
    nmap \
    nmap-scripts

WORKDIR /home/tool
RUN git clone --depth=1 https://github.com/drwetter/testssl.sh.git .
```

Gambar 4.26 Dockerfile *tool*

Dari Dockerfile pada Gambar 4.26, penulis menggunakan basis *image alpine* dengan proses instalasi beberapa *dependencies* yang dibutuhkan untuk menjalankan *tool* testssl.sh kemudian melakukan proses *building* dan *push* kontainer ke kontainer *registry* Docker Hub seperti pada pembahasan implementasi *microservices* tersebut. Gambar 4.27 merupakan potongan format YAML untuk objek konfigurasi pod dari *tool* yang sudah dibuat sebelumnya.

```
apiVersion: v1
kind: Pod
...
  containers:
  - image: riskiwah/ssl-tools
    command:
      - /bin/sh
      - "-c"
      - "sleep 120m"
    imagePullPolicy: Always
    ...
  restartPolicy: Always
```

Gambar 4.27 Pod *tool*

Objek konfigurasi pod pada Gambar 4.27 berjalan dalam satu *namespaces* dengan aplikasi *microservices* yaitu pada namespaces default. Objek konfigurasi pod tersebut berjalan bersamaan secara *sidecar* dengan objek komponen Istio.

4.4.1 Pengujian mutual Trans*port* Layer Security (mTLS)

Pada subbab ini akan membahas tentang pengujian antara *microservices* yang tidak menggunakan *mutual Transport Layer Security* (mTLS) dengan yang menggunakan mTLS. Pertama penulis menggunakan *tool* ksniff untuk melakukan *capture* atau tangkapan komunikasi antara aplikasi *frontend* dan aplikasi *backend* dengan perintah seperti pada Gambar 4.28.

```
~$ kubectl sniff -p -i eth0 -o cap-nonmtls-be.pcap <nama pod> -c istio-
proxy
```

Gambar 4.28 Perintah ksniff

Penjelasan dari perintah pada Gambar 4.28 adalah menjalankan ksniff di atas perintah kubectl dengan keluaran berupa file dengan ekstensi pcap, kemudian untuk kontainernya mengarah ke kontainer istio-proxy (*sidecar* dari Istio). Kemudian langkah

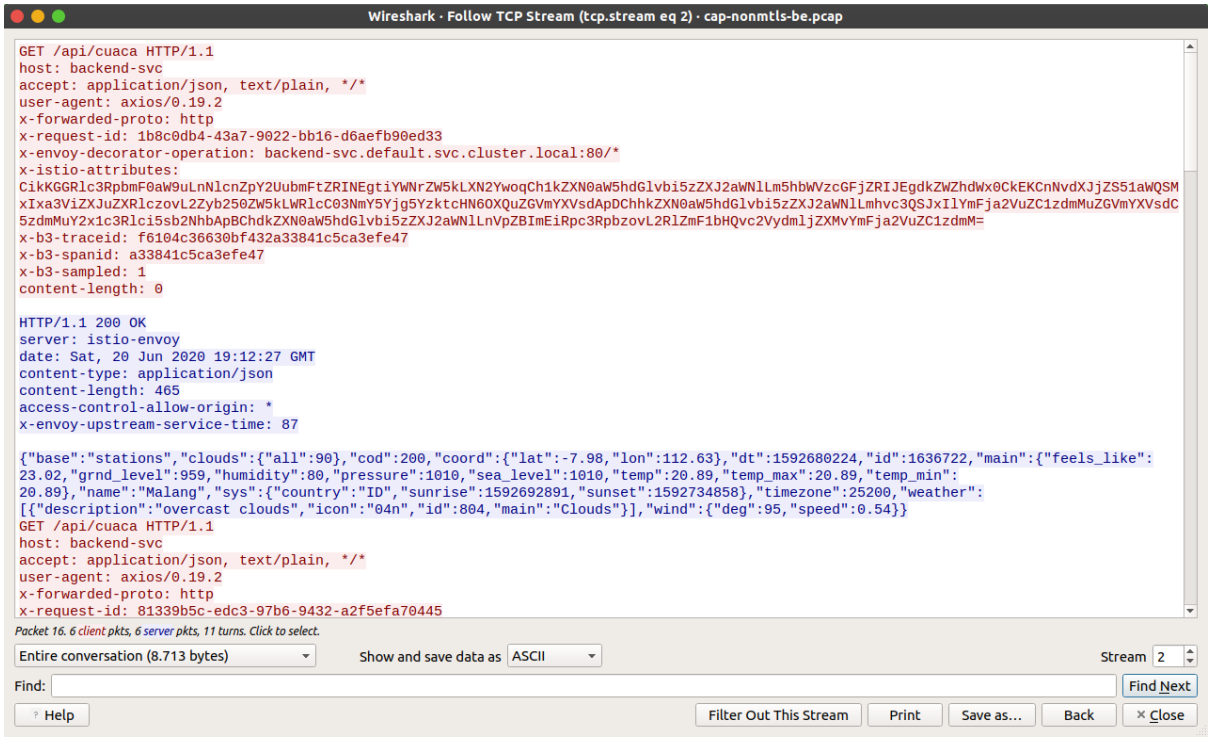
selanjutnya adalah dengan melakukan *generate traffic* ke dalam *microservices* tersebut dengan potongan kode bash seperti pada Gambar 4.29.

```
#!/bin/bash

for ((i=1;i<=10;i++));
do
    curl <ip loadbalancer istio-gateway> > /dev/null 2>&1
    echo "Generate ke $i"
    sleep 2
done
```

Gambar 4.29 Kode Bash *generate traffic*

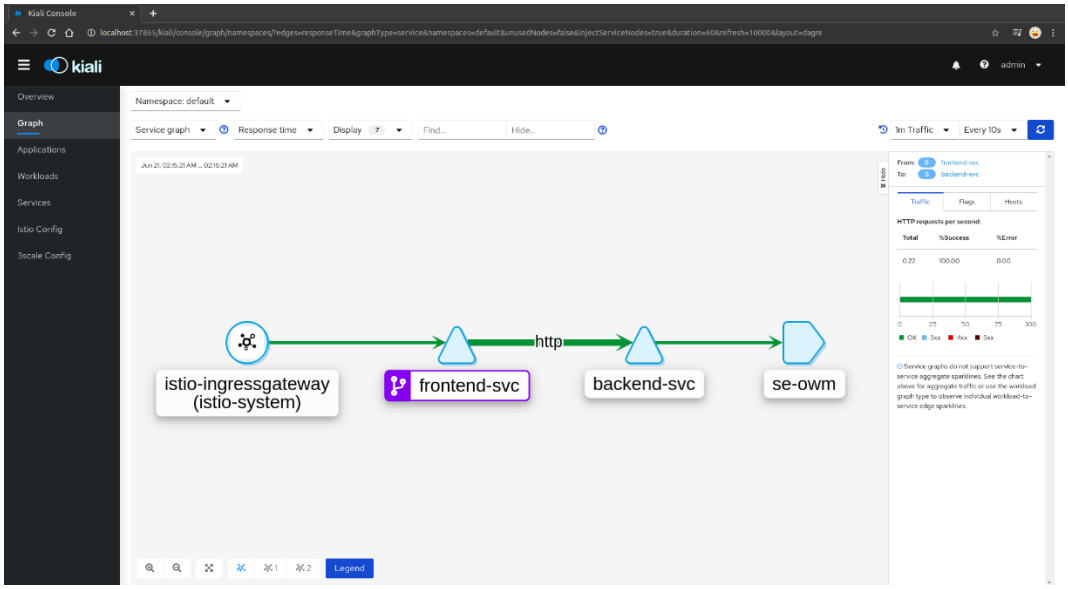
Setelah dari proses *generate traffic*, kemudian membuka file keluaran dengan ekstensi pcap dengan *tool* lanjutan Wireshark dan penulis melakukan penyaringan terhadap *port* yang digunakan oleh aplikasi *backend* yaitu *port* 80 dan penulis melakukan *streaming* pada *packet* dan *port* tersebut seperti Gambar 4.30.



Gambar 4.30 *Streaming* paket tanpa mTLS

Dari Gambar 4.30 diperoleh bahwa ketika *service* atau aplikasi frontend melakukan *request* terhadap *endpoint* /api/cuaca, *request* tersebut dilayani oleh *sidecar* dari Istio, namun karena tidak menggunakan *mutual Transport Layer Security* (mTLS) dalam proses

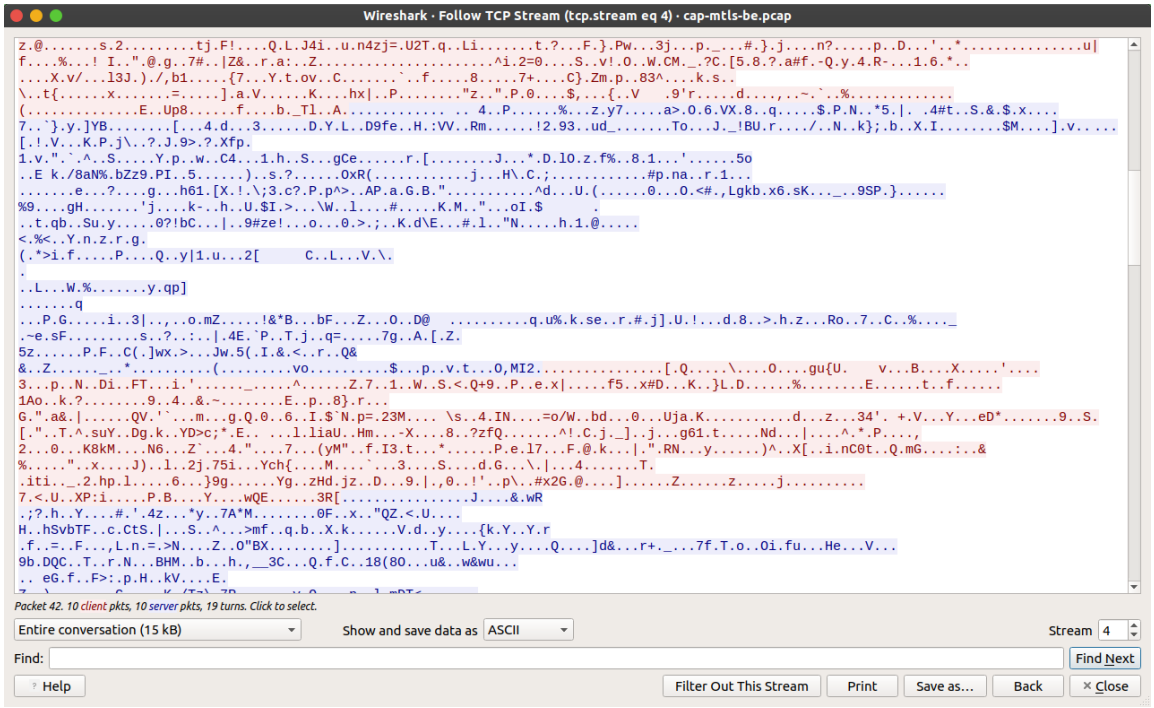
komunikasinya maka data dalam bentuk format JSON tersebut dapat dibaca oleh manusia dengan format ASCII. Kemudian untuk visualisasinya dengan tool Kiali seperti Gambar 4.31.



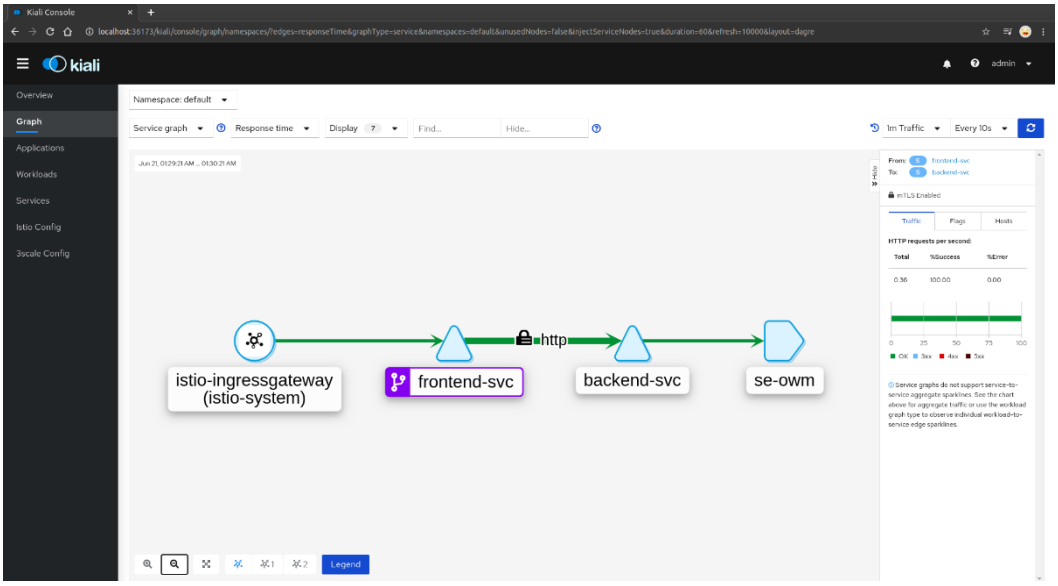
Gambar 4.31 Visualisasi Kiali tanpa mTLS

Dari dua proses pengujian dengan dua *tools* dapat diketahui bahwa *microservices* yang dalam komunikasinya tidak menggunakan *mutual Transport Layer Security* (mTLS) dapat dibaca secara bahasa manusia dan untuk hasil visualisasinya, Kiali tidak mendeteksi adanya komunikasi secara terenkripsi dengan mTLS.

Pengujian selanjutnya adalah *microservices* yang menggunakan *mutual Transport Layer Security* (mTLS) dalam komunikasinya. Hal pertama yang dilakukan penulis adalah sama dengan sebelumnya, yaitu menjalankan *tool* ksniff, melakukan *generate traffic* dan membuka file keluaran berupa ekstensi pcap. Kemudian penulis melakukan penyaringan terhadap *port* 80 dan dibawah ini merupakan hasil dari *streaming* paket dan visualisasi dari *tool* Kiali yang menggunakan mTLS.



Gambar 4.32 *Streaming* paket mTLS



Gambar 4.33 Visualisasi Kiali dengan mTLS

Dari hasil tangkapan layar seperti pada Gambar 4.32 dan Gambar 4.33, hasil dari visualisasi *dashboard* Kiali terdapat gambar gembok yang menandakan komunikasi *microservices* tersebut menggunakan mTLS dan data yang berupa JSON tidak dapat dibaca oleh bahasa manusia, namun dari *tool* Wireshark tersebut tidak dapat melakukan deteksi terhadap protokol yang digunakan oleh SSL/TLS dari Istio. Untuk itu penulis melakukan

pengujian lagi terhadap file paket tersebut dengan beberapa *tool* seperti Ssldump dan Mercury untuk mendeteksi komunikasi sudah menggunakan mTLS. Gambar 4.34 merupakan hasil tangkapan layar untuk analisa lebih lanjut dari dua *tools* tersebut.

```
[290/4316]
riskiwah at extroot-tp: ~/code/dump/final/mtls
➜ cat cap-mtls-be.json | jq 'map(if has("tls") then . else false end) | map(select(. != false))'
[
  {
    "src_ip": "10.0.0.14",
    "dst_ip": "10.0.3.12",
    "src_port": 42606,
    "dst_port": 80,
    "protocol": 6,
    "event_start": 1592677332.170309,
    "fingerprints": {
      "tls": "(0303)(c02bcca9c02fcca8c009c013009c002fc02cc030c00ac014009d0035)((0000)(0017)(ff01)(000a00060004001d0017)(000b00020100)(0023)(0010000
8000605697374696f)(000d001400120403080404010503080505010806060102 01))"
    },
    "tls": {
      "server_name": "outbound_.80_._.backend-svc.default.svc.cluster.local"
    },
    "readable": {
      "tls": {
        "version": "TLS 1.2",
        "cipher_suites": [
          "TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256",
          "TLS_ECDHE_ECDSA_WITH_CHACHA20_POLY1305_SHA256",
          "TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256",
          "TLS_ECDHE_RSA_WITH_CHACHA20_POLY1305_SHA256",
          "TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA",
          "TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA",
          "TLS_RSA_WITH_AES_128_GCM_SHA256",
          "TLS_RSA_WITH_AES_128_CBC_SHA",
          "TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384",
          "TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384",
          "TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA",
          "TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA",
          "TLS_RSA_WITH_AES_256_GCM_SHA384",
          "TLS_RSA_WITH_AES_256_CBC_SHA"
[0] 1:[tmux]*                                                    "extroot-tp" 04:08 23-Jun-20
```

Gambar 4.34 Tangkap layar *tool* Mercury

Hasil dari penggunaan *tool* Mercury adalah berupa stdout JSON, kemudian penulis melakukan seleksi setiap *object* dari JSON yang terdapat string TLS. Tool Mercury dapat mengidentifikasi bahwa terdapat komunikasi yang menggunakan SSL/TLS dengan beberapa parameter dalam bentuk *object property* JSON seperti sumber dan tujuan paket kemudian versi SSL/TLS yang digunakan serta *cipher* dan algoritma kriptografi yang digunakan.

```
New TCP connection #4: 10.0.0.14(42606) <-> 10.0.3.12(80)
4 1  0.0006 (0.0006)  C>SV3.1(200)  Handshake      ClientHello
        random[32]=
          63 c9 b4 b2 ba 26 f8 d9 51 c6 07 2c 15 47 b2 95
          09 c3 e7 73 7f 8f 80 86 68 80 39 a1 57 20 e8 6c
4 2  0.0026 (0.0020)  S>CV3.3(63)  Handshake      ServerHello
        random[32]=
          5e ee 53 d4 94 39 c3 e0 61 4c e6 c6 65 32 5d 86
          ba b9 40 d8 22 cc 7a 5b 44 4f 57 4e 47 52 44 01
4 3  0.0026 (0.0000)  S>CV3.3(811)  Handshake      Certificate
4 4  0.0026 (0.0000)  S>CV3.3(300)  Handshake      ServerKeyExchange
4 5  0.0026 (0.0000)  S>CV3.3(57)  Handshake      CertificateRequest
Not enough data. Found 46 bytes (expecting 32767)
4 6  0.0026 (0.0000)  S>CV3.3(4)  Handshake      ServerHelloDone
4 7  0.0047 (0.0020)  C>SV3.3(811)  Handshake      Certificate
4 8  0.0047 (0.0000)  C>SV3.3(37)  Handshake      ClientKeyExchange
4 9  0.0047 (0.0000)  C>SV3.3(264)  Handshake      CertificateVerify
Not enough data. Found 258 bytes (expecting 16384)
4 10 0.0047 (0.0000)  C>SV3.3(1)  ChangeCipherSpec
4 11 0.0047 (0.0000)  C>SV3.3(40)  Handshake
4 12 0.0051 (0.0004)  S>CV3.3(1018)  Handshake4 13 0.0051 (0.0000)  S>CV3.3(1)  ChangeCipherSpec
4 14 0.0051 (0.0000)  S>CV3.3(40)  Handshake
4 15 0.0053 (0.0001)  C>SV3.3(814)  application_data
```

Gambar 4.35 Tangkap layar *tool* Ssldump

Dari tangkapan layar pada Gambar 4.35 yang menggunakan *tool* Ssldump, penulis mendapatkan hasil bahwa komunikasi SSL/TLS dilakukan secara dua arah atau *mutual*. Hasil tersebut diperoleh pada keluaran *tool* Ssldump yang berupa stdout kolom enam sampai dengan tujuh. Setelah dari hasil pengujian dan pembuktian tersebut, penulis melakukan pengecekan terhadap *Public Key Infrastructure* (PKI) yang dimiliki oleh Istio dengan bantuan pod yang sudah dibuat sebelumnya. Beberapa keluaran yang dapat dijadikan informasi adalah dari Istio *port* atau akses yang dibuka untuk melakukan tugas sebagai *Public Key Infrastructure* (PKI) adalah Istio Pilot dan dapat mendukung sampai dengan TLS versi 1.3.

Istio Pilot sendiri memiliki nilai *validity period certificate* selama 60 hari dengan *cert issuer* adalah cluster.local sesuai dengan bawaan default DNS dari infrastruktur Kubernetes. Kemudian *field* untuk *Subject Alternative Name* berupa url dengan format spiffe://<issuer>/ns/<namespace>/sa/<*service*account>.

4.4.2 Pengujian Performa Micro*services*

Pada sub bab terakhir ini merupakan hasil pengujian performa dari *microservices* yang sudah dibuat sebelumnya dengan perbandingan performa *microservices* yang berkomunikasi tanpa *mutual Transport Layer Security* (mTLS) dengan yang sudah menggunakan mTLS. Pengujian tersebut dilakukan dengan melakukan request method POST dari API menuju S3 server yang besaran ukuran file upload dilakukan secara dan untuk pengujiannya

menggunakan *tool* Apache Jmeter. Gambar 4.36 merupakan hasil dari *microservices* yang belum menggunakan mTLS.

Statistics

Requests	Executions			Response Times (ms)							Throughput	Network (KB/sec)	
Label	#Samples	KO	Error %	Average	Min	Max	Median	90th pct	95th pct	99th pct	Transactions/s	Received	Sent
Total	**100**	**0**	**0.00%**	**3414.55**	**173**	**21003**	**2049.50**	**6464.70**	**19002.85**	**20995.99**	**0.15**	**0.03**	**192.29**
POST-UPLOAD	100	0	0.00%	3414.55	173	21003	2049.50	6464.70	19002.85	20995.99	0.15	0.03	192.29

Gambar 4.36 Hasil tanpa mTLS

Dari Gambar 4.36, *sample request* yang digunakan berjumlah 100 *request* dengan nilai *error rate* sejumlah 0% dan untu rata-rata *response time* adalah 3414,55 milidetik atau sekitar 3,41455 detik. Kemudian untuk nilai besaran nilai transfer data (*throughput*) adalah 0,15 dengan total waktu pengujian selama 11 menit. Selanjutnya Gambar 4.37 merupakan hasil dari pengujian *microservices* yang sudah menggunakan mTLS.

Statistics

Requests	Executions			Response Times (ms)							Throughput	Network (KB/sec)	
Label	#Samples	KO	Error %	Average	Min	Max	Median	90th pct	95th pct	99th pct	Transactions/s	Received	Sent
Total	**100**	**0**	**0.00%**	**5255.99**	**35**	**34900**	**2100.50**	**21069.00**	**34687.10**	**34899.99**	**0.16**	**0.03**	**220.79**
POST-UPLOAD	100	0	0.00%	5255.99	35	34900	2100.50	21069.00	34687.10	34899.99	0.16	0.03	220.79

Gambar 4.37 Hasil dengan mTLS

Sama seperti pada pengujian sebelumnya menggunakan 100 *request* dengan hasil memiliki nilai *error rate* 0% dan untuk nilai rata-rata dari *response time* 5255,99 milidetik atau sekitar 5,25599 detik. Kemudian untuk besaran nilai transfer data adalah 0,16 dengan waktu pengujian selama 10 menit.

BAB V
KESIMPULAN DAN SARAN

5.1 Kesimpulan

Berdasarkan hasil penelitian yang telah dibahas sebelumnya tentang implementasi *mutual Transport Layer Security* (mTLS) pada arsitektur *microservices*, dapat ditarik beberapa kesimpulan yang mengacu pada tujuan dari penelitian ini, di antaranya adalah:

a. Implementasi *mutual Transport Layer Security* (mTLS) dapat diterapkan pada arsitektur *microservices* yang berjalan di atas infrastruktur Kubernetes dengan memanfaatkan fitur keamanan dari *service mesh* pada *tool* Istio. Aplikasi atau *services* yang berjalan berkomunikasi secara dua arah menggunakan protokol SSL/TLS dan setiap *services* tersebut memiliki sertifikat digital dan melakukan proses *handshake* dengan verifikasi dari *Public Key Infrastructure* (PKI) yang dilakukan oleh dua *control plane* dari Istio citadel dan Istio pilot.

b. Dengan menggunakan beberapa *tools* kita dapat mengetahui komunikasi *microservices* yang sudah menggunakan *mutual Transport Layer Security* (mTLS) dengan yang tidak menggunakan mTLS. *Tools* pertama yaitu Wireshark memberikan kita informasi bahwa paket data yang tidak menggunakan mTLS dapat dibaca langsung oleh manusia sedangkan komunikasi *microservices* yang menggunakan mTLS tidak dapat dibaca oleh manusia karena sudah terenkripsi dengan standar protokol kriptografi TLS. Kemudian hasil kesimpulan dari keluaran *tools* Mercury dan ssldump membuktikan bahwa terjadi komunikasi *microservices* yang menggunakan autentikasi dua arah dengan versi kriptografi Trans*port* Layer Security (TLS) versi 1.2, sedangkan menurut *tool* testssl mampu mendukung sampai dengan TLS versi 1.3.

c. Dari hasil perbandingan pengujian performa yang sudah dilakukan dengan *tool* Apache Jmeter. Untuk parameter *error rate*, *microservices* yang tidak menggunakan komunikasi *mutual Transport Layer Security* (mTLS) kedua perbandingan pengujian performa tersebut sama memiliki nilai error rate 0%. Kemudian untuk parameter *response time microservices* yang tidak menggunakan autentikasi dua arah lebih cepat 1841,44 milidetik daripada *microservices* yang menggunakan autentikasi dua arah. Perbedaan waktu hampir 2 detik tersebut terjadi karena proses mTLS membutuhkan waktu untuk melakukan proses *handshake* pertukaran sertifikat, kunci dan proses enkripsi dekripsi.

5.2 Saran

Pada penelitian ini ditemukan beberapa kendala-kendala dan juga kekurangan, saran untuk pengembangan maupun penelitian serupa adalah:

a. Penerapan metode pengamanan *microservices* selain menggunakan *mutual Transport Layer Security* (mTLS).
b. Implementasi berbagai skenario pengujian yang berbeda terhadap komunikasi *microservices* yang menggunakan autentikasi dua arah, selain dari skenario juga dalam hal *tools* maupun jenis *service mesh* yang akan digunakan.
c. Implementasi infrastruktur Kubernetes yang lebih mudah, ringan dan mungkin tidak memakan biaya.

DAFTAR PUSTAKA

Alkhulaifi, A., & El-Alfy, E. S. M. (2020). Exploring Lattice-based Post-Quantum Signature for JWT Authentication: Review and Case Study. *IEEE Vehicular Technology Conference*, *2020-May*, 2–6. https://doi.org/10.1109/VTC2020-Spring48590.2020.9129505

Arundel, J., & Domingus, J. (2019). Cloud Native DevOps with Kubernetes. In *O'Reilly Media, Inc*.

Burns, B., Beda, J., & Hightower, K. (2019). *Kubernetes up & running*.

Calcote, L. (2018). *The Enterprise Path to Service Mesh Architectures*.

Calcote, L., & Butcher, Z. (2020). *Istio: Up & Running*.

Coelho, N. M. (2018). *Security in Microservices Architectures*. *December*. http://colinmorelli.com/blog/2015/5/21/security-in-microservices

developers@vub.sk. (2019). *PSD2 documentation API Documentation for Third party providers*. *1.1*, 1–59. https://www.vub.sk/files/firmy-podnikatelia/produkty-sluzby/platby/ini-poskytovatelia-sluzieb/vub_psd2_documentation_2019-05-30.pdf

Garrett, O. (2019). *Do I Need a Service Mesh? - NGINX*. https://www.nginx.com/blog/do-i-need-a-service-mesh/

Groskop, M., Pariente, N., Scialabba, L. and Smith, D. (2020). *PROTECTING WHAT YOU CAN ' T SEE*.

Indrasiri, K., & Siriwardena, P. (2018). *Microservices for the Enterprise: Designing, Developing, and Deploying*.

Istio Authors. (2019). *Istioldie 1.4 / Security*. https://archive.istio.io/v1.4/docs/concepts/security/#pki

Jones, M., Bradley, J., & Sakimura, N. (2015). RFC 7519: Json web token (JWT). In *Internet Engineering Task Force (IETF)* (p. 30). https://doi.org/2070-1721

Kang, M., Shin, J. S., & Kim, J. (2019). Protected Coordination of Service Mesh for Container-Based 3-Tier Service Traffic. *International Conference on Information Networking*, *2019-Janua*, 427–429. https://doi.org/10.1109/ICOIN.2019.8718120

Manpathak, S. (2019). *Kubernetes Service Mesh: A Comparison of Istio, Linkerd and Consul*. https://platform9.com/blog/kubernetes-service-mesh-a-comparison-of-istio-linkerd-and-consul/

Monahan, D. (2019). *THE STATE OF WEB APPLICATION Protecting Applications in the*

Microservice Era Table of Contents.

Munir, R. (2019). *Kriptografi Edisi Kedua* (2nd ed.). INFORMATIKA.

Palm, J. (2018). *Service isolation in large microservice networks*.

Siriwardena, P. (2020). Advanced API Security. In *Advanced API Security*. https://doi.org/10.1007/978-1-4842-2050-4

Suomalainen, J. (2019). *Defense-in-Depth Methods in Microservices Access Control*. https://trepo.tuni.fi/handle/123456789/27172

Sutter, C. P. B. (2018). *Introducing Istio Service Mesh for Microservices*.

The Economist Newspaper. (2015). *The cost of immaturity*. https://www.economist.com/business/2015/11/05/the-cost-of-immaturity

The Kubernetes Authors. (2020). *Namespaces | Kubernetes*. https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces/

Wolff, E., & Prinz, H. (2020). *Service Mesh - The New Infrastructure for Microservices*. innoQ Deutschland GmbH.

WSO2 Inc. (2020). *API Security - WSO2 Open Banking 1.3.0 - WSO2 Documentation*. https://docs.wso2.com/display/OB130/API+Security#APISecurity-MutualTransportLayerSecurity(MTLS)

Yarygina, T., & Bagge, A. H. (2018). Overcoming Security Challenges in Microservice Architectures. *Proceedings - 12th IEEE International Symposium on Service-Oriented System Engineering, SOSE 2018 and 9th International Workshop on Joint Cloud Computing, JCC 2018*, 11–20. https://doi.org/10.1109/SOSE.2018.00011

LAMPIRAN

Untuk lampiran penuh dapat dilihat pada tautan berikut.

https://github.com/riskiwah/istio-mtls

Cap.txt


```
[
  {
    "src_ip": "10.0.0.14",
    "dst_ip": "10.0.3.12",
    "src_port": 42606,
    "dst_port": 80,
    "protocol": 6,
    "event_start": 1592677332.170309,
    "fingerprints": {
      "tls":
"(0303)(c02bcca9c02fcca8c009c013009c002fc02cc030c00ac014009d0035)((0000)(0017)(ff
01)(000a00060004001d0017)(000b00020100)(0023)(00100008000605697374696f)(000d001
400120403080404010503080505010806060102 01))"
    },
    "tls": {
      "server_name": "outbound_.80_._.backend-svc.default.svc.cluster.local"
    },
    "readable": {
      "tls": {
        "version": "TLS 1.2",
        "cipher_suites": [
          "TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256",
          "TLS_ECDHE_ECDSA_WITH_CHACHA20_POLY1305_SHA256",
          "TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256",
          "TLS_ECDHE_RSA_WITH_CHACHA20_POLY1305_SHA256",
          "TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA",
          "TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA",
          "TLS_RSA_WITH_AES_128_GCM_SHA256",
          "TLS_RSA_WITH_AES_128_CBC_SHA",
          "TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384",
          "TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384",
          "TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA",
          "TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA",
          "TLS_RSA_WITH_AES_256_GCM_SHA384",
          "TLS_RSA_WITH_AES_256_CBC_SHA"
        ],
        "extensions": [
          {
            "server_name": ""
          },
```

```
      {
        "extended_master_secret": ""
      },
      {
        "renegotiation_info": ""
      },
      {
        "supported_groups": {
          "supported_groups_list_length": 4,
          "supported_groups": [
            "x25519",
            "secp256r1"
          ]
        }
      },
      {
        "ec_point_formats": {
          "ec_point_formats_length": 1,
          "ec_point_formats": [
            "uncompressed"
          ]
        }
      },
      {
        "session_ticket": ""
      },
      {
        "application_layer_protocol_negotiation": [
          "istio"
        ]
      },
      {
        "signature_algorithms": {
          "signature_hash_algorithms_length": 18,
          "algorithms": [
            "ecdsa_sha256",
            "rsa_pss_sha256",
            "rsa_sha256",
            "ecdsa_sha384",
            "rsa_pss_sha384",
            "rsa_sha384",
            "rsa_pss_sha512",
            "rsa_sha512",
            "rsa_sha1"
          ]
        }
      }
    ]
  }
```

```
    },
    "analysis": {
      "process": "Firefox",
      "score": 1,
      "category": "browser"
    }
  },
  {
    "src_ip": "10.0.3.12",
    "dst_ip": "10.0.0.14",
    "src_port": 80,
    "dst_port": 42606,
    "protocol": 6,
    "event_start": 1592677332.172309,
    "fingerprints": {
      "tls_server": "(0303)(c02f)((0017)(ff01)(000b00020100)(0023))"
    },
    "tls": {
      "server_certs": [

"MIIDHTCCAgWgAwIBAgIQTvSwKL4csO7Jty6EHLo67DANBgkqhkiG9w0BAQsFADA
YMRYwFAYDVQQKEw1jbHVzdGVyLmxvY2FsMB4XDTIwMDYyMDEwMjkzM1oXD
TIwMDkxODEwMjkzM1owADCCASIwDQYJKoZIhvcNAQEBBQADggEPADCCAQoCg
gEBAL0nedGs+Tp8W6kXYrUjJCI4FPfhTH6tdpsGHKAA9GerM35GzbQVuPVmtdWPF/E
IC4nJXQn77wrCogjjjLFe4FfivwLyo6qYusHiRVJ4kB1F8DK3KUIoY1DRO6I72OqrtRtpn0
CxUySWsZ/Xd/omOE3qRH1Lt+ePf7XtBj/mxZvhDEtN6xjdxSowd29NWnqMpPT1dX5DnI
KBQPTAii1KW6GxzNBap0lo/oaE415DIH235868Xh79szC03Rizitck8VJFkLqY36uqxhOdf
AR7otegYr3ft2f7PQwm2uvwOeaRBQ+yu1MMtVYnuu79CGUNCRNiNwuF1D1fZ4arvjo
W1GUCAwEAAaN7MHkwDgYDVR0PAQH/BAQDAgWgMB0GA1UdJQQWMBQGCCs
GAQUFBwMBBggrBgEFBQcDAjAMBgNVHRMBAf8EAjAAMDoGA1UdEQEB/wQwM
C6GLHNwaWZmZTovL2NsdXN0ZXIubG9jYWwvbnMvZGVmYXVsdC9zYS9kZWZhd
Wx0MA0GCSqGSIb3DQEBCwUAA4IBAQBMQGnB2KKOuFn2oBboEKMGLk3/H5ogpE
xNqzNafajUZy1E5+JxTcbl/4XosISo1sHMTEy+Bo0vYDtPLRZpuc6xKb6j9oKFI5BWQLS
AZLv6iR5GDqqRYZTq/iL9Hyi3FGqtMdCXDlg059lTImRrgkDa6D9SAvyPw8IhetNAyL7n
shcu0nPPMv2zoa2Q/Nrvo3RqEkYh/MQuAFHjTBRKNGkFoHUIbjR6aj34VTJU6XGM5Ex
p/fvToq3YHXTkP6W9f0aUfX9Qd9ucvzNqEafscLRfic77I+x9CGqExKKMbj+2zofZ63CF3
kTEvJQn7RYqxpDUH/CLG8quoszhob3ldXxm"
      ]
    },
    "readable": {
      "tls_server": {
        "version": "TLS 1.2",
        "selected_cipher_suite": [
          "TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256"
        ],
        "extensions": [
          {
            "extended_master_secret": ""
          },
```

```
        {
          "renegotiation_info": ""
        },
        {
          "ec_point_formats": {
            "ec_point_formats_length": 1,
            "ec_point_formats": [
              "uncompressed"
            ]
          }
        },
        {
          "session_ticket": ""
        }
      ]
    },
    "server_certs": [
      {
        "version": "v3",
        "serial_number": "4ef4b028be1cb0eec9b72e841cba3aec",
        "issuer": [
          {
            "id-at-organizationName": "cluster.local"
          }
        ],
        "validity": {
          "not_before": "200620102933Z",
          "not_after": "200918102933Z"
        },
        "subject": [],
        "extensions": [
          {
            "id-ce-keyUsage": "ff"
          },
          {
            "id-ce-extKeyUsage": "301406082b0601050507030106082b06010505070302"
          },
          {
            "id-ce-basicConstraints": "ff"
          },
          {
            "id-ce-subjectAltName": "ff"
          }
        ]
      }
    ]
  }
},
{
```

"src_ip": "10.0.0.14",
"dst_ip": "10.0.3.12",
"src_port": 42606,
"dst_port": 80,
"protocol": 6,
"event_start": 1592677332.174389,
"tls": {
"server_certs": [

"MIIDHTCCAgWgAwIBAgIQTvSwKL4csO7Jty6EHLo67DANBgkqhkiG9w0BAQsFADA
YMRYwFAYDVQQKEw1jbHVzdGVyLmxvY2FsMB4XDTIwMDYyMDEwMjkzM1oXD
TIwMDkxODEwMjkzM1owADCCASIwDQYJKoZIhvcNAQEBBQADggEPADCCAQoCg
gEBAL0nedGs+Tp8W6kXYrUjJCI4FPfhTH6tdpsGHKAA9GerM35GzbQVuPVmtdWPF/E
IC4nJXQn77wrCogjjjLFe4FfivwLyo6qYusHiRVJ4kB1F8DK3KUIoY1DRO6I72OqrtRtpn0
CxUySWsZ/Xd/omOE3qRH1Lt+ePf7XtBj/mxZvhDEtN6xjdxSowd29NWnqMpPT1dX5DnI
KBQPTAii1KW6GxzNBap0lo/oaE415DIH235868Xh79szC03Rizitck8VJFkLqY36uqxhOdf
AR7otegYr3ft2f7PQwm2uvwOeaRBQ+yu1MMtVYnuu79CGUNCRNiNwuF1D1fZ4arvjo
W1GUCAwEAAaN7MHkwDgYDVR0PAQH/BAQDAgWgMB0GA1UdJQQWMBQGCCs
GAQUFBwMBBggrBgEFBQcDAjAMBgNVHRMBAf8EAjAAMDoGA1UdEQEB/wQwM
C6GLHNwaWZmZTovL2NsdXN0ZXIubG9jYWwvbnMvZGVmYXVsdC9zYS9kZWZhd
Wx0MA0GCSqGSIb3DQEBCwUAA4IBAQBMQGnB2KKOuFn2oBboEKMGLk3/H5ogpE
xNqzNafajUZy1E5+JxTcbl/4XosISo1sHMTEy+Bo0vYDtPLRZpuc6xKb6j9oKFI5BWQLS
AZLv6iR5GDqqRYZTq/iL9Hyi3FGqtMdCXDlg059lTImRrgkDa6D9SAvyPw8IhetNAyL7n
shcu0nPPMv2zoa2Q/Nrvo3RqEkYh/MQuAFHjTBRKNGkFoHUIbjR6aj34VTJU6XGM5Ex
p/fvToq3YHXTkP6W9f0aUfX9Qd9ucvzNqEafscLRfic77I+x9CGqExKKMbj+2zofZ63CF3
kTEvJQn7RYqxpDUH/CLG8quoszhob3ldXxm"
]
},
"readable": {
"server_certs": [
{
"version": "v3",
"serial_number": "4ef4b028be1cb0eec9b72e841cba3aec",
"issuer": [
{
"id-at-organizationName": "cluster.local"
}
],
"validity": {
"not_before": "200620102933Z",
"not_after": "200918102933Z"
},
"subject": [],
"extensions": [
{
"id-ce-keyUsage": "ff"
},
{
"id-ce-extKeyUsage": "301406082b0601050507030106082b06010505070302"

```
          },
          {
            "id-ce-basicConstraints": "ff"
          },
          {
            "id-ce-subjectAltName": "ff"
          }
        ]
      }
    ]
  }
},
{
  "src_ip": "10.0.0.14",
  "dst_ip": "10.0.3.12",
  "src_port": 42728,
  "dst_port": 80,
  "protocol": 6,
  "event_start": 1592677350.463189,
  "fingerprints": {
    "tls":
"(0303)(c02bcca9c02fcca8c009c013009c002fc02cc030c00ac014009d0035)((0000)(0017)(ff
01)(000a00060004001d0017)(000b00020100)(0023)(0010000800060569737469 6f)(000d001
400120403080404010503080505010806060102 01))"
  },
  "tls": {
    "server_name": "outbound_.80_._.backend-svc.default.svc.cluster.local"
  },
  "readable": {
    "tls": {
      "version": "TLS 1.2",
      "cipher_suites": [
        "TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256",
        "TLS_ECDHE_ECDSA_WITH_CHACHA20_POLY1305_SHA256",
        "TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256",
        "TLS_ECDHE_RSA_WITH_CHACHA20_POLY1305_SHA256",
        "TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA",
        "TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA",
        "TLS_RSA_WITH_AES_128_GCM_SHA256",
        "TLS_RSA_WITH_AES_128_CBC_SHA",
        "TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384",
        "TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384",
        "TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA",
        "TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA",
        "TLS_RSA_WITH_AES_256_GCM_SHA384",
        "TLS_RSA_WITH_AES_256_CBC_SHA"
      ],
      "extensions": [
        {
```

```
        "server_name": ""
      },
      {
        "extended_master_secret": ""
      },
      {
        "renegotiation_info": ""
      },
      {
        "supported_groups": {
          "supported_groups_list_length": 4,
          "supported_groups": [
            "x25519",
            "secp256r1"
          ]
        }
      },
      {
        "ec_point_formats": {
          "ec_point_formats_length": 1,
          "ec_point_formats": [
            "uncompressed"
          ]
        }
      },
      {
        "session_ticket": ""
      },
      {
        "application_layer_protocol_negotiation": [
          "istio"
        ]
      },
      {
        "signature_algorithms": {
          "signature_hash_algorithms_length": 18,
          "algorithms": [
            "ecdsa_sha256",
            "rsa_pss_sha256",
            "rsa_sha256",
            "ecdsa_sha384",
            "rsa_pss_sha384",
            "rsa_sha384",
            "rsa_pss_sha512",
            "rsa_sha512",
            "rsa_sha1"
          ]
        }
      }
```

```
      ]
     }
    },
    "analysis": {
     "process": "Firefox",
     "score": 1,
     "category": "browser"
    }
  }
]
```