2021

PEDOMAN KEAMANAN

MICROSERVICE DAN APPLICATION PROGRAMMING INTERFACE (API)

Berdasarkan:

NIST SP 800-204 (2019)

OWASP API Security Top 10 (2019)

PUSAT PENGKAJIAN DAN PENGEMBANGAN
TEKNOLOGI KEAMANAN SIBER DAN SANDI

Pedoman Keamanan

Microservice dan *Application Programming Interface* (API)

Pengarah : Anton Setiyawan

Tim Penyusun:

1. Adrian Admi
2. Feri Ramdani
3. Wahyu Achmad Fadiel
4. Mohamad Ali Sadikin
5. Jonatan Reky Tasyam
6. Abdul Azzam Ajhari
7. Wildan Hilmy

Penyunting Bahasa: Bety Mawarni

Badan Siber dan Sandi Negara (BSSN)
Jl. Harsono RM 70 Ragunan
Pasar Minggu, Jakarta Selatan, 12550
Tel: +62217805814
Fax: +622178844104
Email: humas@bssn.go.id &
https://www.bssn.go.id

Versi 1.0 | Agustus 2021 | Badan Siber dan Sandi Negara

REDAKSI KITA

Microservice sudah menjadi bagian dari kebanyakan sistem aplikasi yang dikembangkan saat ini. Dengan adanya *microservice*, memudahkan setiap pengembang dalam membangun sebuah aplikasi sehingga menjadi lebih sederhana dan fleksibel.

Sayangnya, dengan banyaknya penggunaan *microservice* saat ini, kerap kali terdapat celah di dalamnya yang digunakan oleh para *hacker* untuk menyusup ke suatu sistem aplikasi dan mengambil data yang ada pada basis data. Oleh karena itu, walaupun memiliki beberapa kelebihan dalam membangun sebuah sistem aplikasi berbasis *microservice,* para pengembang aplikasi juga perlu mempertimbangkan strategi keamanan *microservice* yang digunakan.

Terdapat dua contoh kasus insiden keamanan API yang terjadi pada tahun 2018, yang pertama terjadi pada web aplikasi penyedia layanan manajemen repositori Gitlab, yaitu adanya kerentanan pada Gitlab Event API yang dapat digunakan untuk mengekspos informasi rahasia pada *project* kode[1]. Yang kedua terjadi pada web aplikasi media sosial Facebook yaitu terdapat kerentanan pada API pengembang Facebook yang digunakan oleh *hacker* untuk dapat mengekspos jutaan penggunanya[2].

Selain kasus-kasus tersebut, masih banyak lagi kasus terkait insiden keamanan API yang pernah terjadi di antaranya adalah API eksploit pada Twitter yang digunakan untuk mengekspos nomor telepon pengguna[3], kerentanan pada API *sign-in* milik Apple yang memungkinkan pengguna untuk *login* sebagai orang lain[4], kerentanan API pada VMware yang dapat mengambil alih akun admin[5], dan masih banyak lagi.

Dengan maraknya kasus insiden keamanan API tersebut, maka BSSN berinisiatif menerbitkan pedoman dalam mengembangkan aplikasi berbasis *microservice* dan API yang aman. Besar harapan kami semoga pedoman ini dapat digunakan sebagai rujukan oleh para pengembang dan pimpinan pada bagian Teknologi Informasi di perusahaan swasta maupun pemerintahan.

Salam,

Anton Setiyawan

1. https://latesthackingnews.com/2018/10/08/gitlab-api-vulnerability-leaked-confidential-data-on-public-projects/.
2. https://techcrunch.com/2018/09/28/everything-you-need-to-know-about-facebooks-data-breach-affecting-50m-users/.
3. https://privacy.twitter.com/en/blog/2020/an-incident-impacting-your-account-identity
4. https://latesthackingnews.com/2020/05/31/researcher-discovers-critical-vulnerability-and-was-awarded-100000/
5. https://citadelo.com/en/blog/full-infrastructure-takeover-of-vmware-cloud-director-CVE-2020-3956/

DAFTAR ISI

REDAKSI KITA ..... **iii**
DAFTAR ISI ..... **iv**
FENOMENA **1**
1. PENDAHULUAN **3**
1.1. Ruang Lingkup 3
1.2. Target Audiensi 4
1.3. Rujukan Penting Lainnya 4
1.4. Sistematika Penyusunan Dokumen 4
2. LATAR BELAKANG TEKNOLOGI **5**
2.1. Konsep Dasar 5
2.2. Prinsip Desain 5
2.3. Faktor Penggerak Bisnis 6
2.4. Blok Pembangun (*Building Block*) 7
2.5. Gaya Interaksi 8
2.6. Praktik Fitur-fitur Inti 10
2.7. Kerangka Arsitektur 12
2.7.1. API *Gateway* 13
2.7.2. *Service Mesh* 15
2.8. Perbandingan dengan Arsitektur Monolitik 16
2.9. Perbandingan dengan *Service-Oriented Architecture* (SOA) 16
2.10. Keuntungan *Microservice* 17
2.11. Kekurangan *Microservice* 18
3. LATAR BELAKANG ANCAMAN **19**
3.1. Ulasan Sumber Ancaman 19
3.2. Ancaman Khusus Pada *Microservice* 20
3.2.1. Ancaman Mekanisme *Service Discovery* 20
3.2.2. Serangan Berbasis Internet 21
3.2.3. Kegagalan berjenjang 22
4. STRATEGI KEAMANAN UNTUK MENERAPKAN FITUR INTI DAN MELAWAN ANCAMAN **23**
4.1. Strategi untuk Manajemen Identitas dan Manajemen Akses 23
Strategi Keamanan untuk Autentikasi (Strategi 1) 24
Strategi Keamanan untuk Manajemen Akses (Strategi 2) 25
4.2. Strategi untuk Mekanisme *Service discovery* 25
Strategi Keamanan untuk Konfigurasi *Service Registry* (Strategi 3) 28
4.3. Strategi untuk Protokol Komunikasi yang Aman 28
Strategi Keamanan untuk Komunikasi Yang Aman (Strategi 4) 29

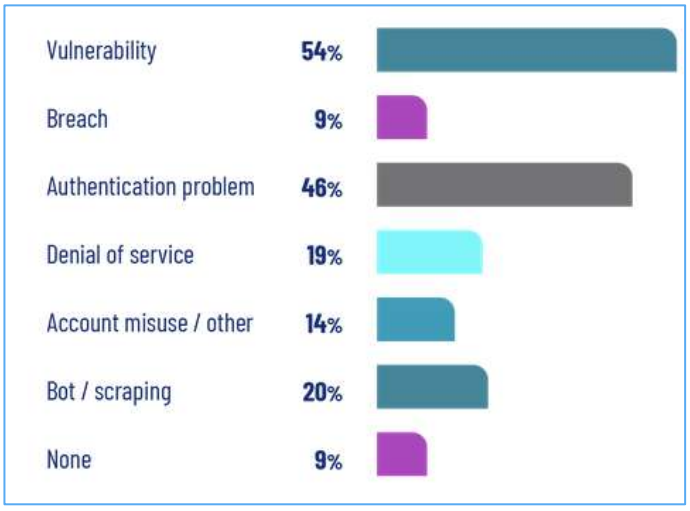
4.4. Strategi untuk Pemantauan Keamanan (*Security Monitoring*) 29
Strategi Keamanan untuk Pemantauan Keamanan (Strategi 5) 30
4.5. Strategi Peningkatan Ketersediaan / Ketahanan 30
4.5.1. Analisis opsi implementasi Pemutus sirkuit 30
Strategi Keamanan untuk Menerapkan Pemutus Sirkuit (Strategi 6) 32
4.5.2. Strategi untuk *Load Balancing* 32
Strategi Keamanan untuk *Load Balancing* (Strategi 7) 32
4.5.3. Pembatasan Laju 32
Strategi Keamanan untuk Pembatasan Laju (Strategi 8) 33
4.6. Strategi Penjaminan Integritas 33
Strategi Keamanan (Jaminan Integritas) untuk Implementasi *Microservice* Versi Baru (Strategi 9) 34
Strategi Keamanan untuk Menangani Persistensi Sesi (Strategi 10) 35
4.7. Melawan Serangan Berbasis Internet 35
Strategi Keamanan Mencegah *Credential Abuse* dan *Stuffing Attacks* (Strategi 11) 35
5. STRATEGI KEAMANAN UNTUK KERANGKA ARSITEKTUR *MICROSERVICE* 36
Strategi Keamanan untuk Implementasi API *Gateway* (Strategi 12) 36
Strategi Keamanan untuk Implementasi *Service Mesh* (Strategi 13) 37
LAMPIRAN A - PERBEDAAN ANTARA APLIKASI MONOLITIK DAN APLIKASI BERBASIS *MICROSERVICE* 38
A.1. Perbedaan Desain dan Penerapan 38
A.1.1. Contoh Aplikasi untuk Mengilustrasikan Perbedaan Desain dan Penerapan 39
A.2. Perbedaan Selama Operasi 41
LAMPIRAN B - PENELUSURAN STRATEGI KEAMANAN FITUR ARSITEKTUR *MICROSERVICE* 43
LAMPIRAN C - OWASP API *SECURITY* TOP 10 (2019) 51
CATATAN UNTUK PENGEMBANG 82
CATATAN UNTUK DEV-SEC-OPS 83

FENOMENA

Perusahaan asal Amerika Serikat di bidang keamanan API, Salt Security, telah merilis laporan keamanan API-nya yang berjudul, "The State of API Security – Q1 2021". Untuk membuat laporan ini, Salt Security mengumpulkan data pelanggan anonim dan tanggapan survei dari sekitar 200 profesional keamanan, aplikasi dan DevOps. Responden survei berasal dari berbagai perusahaan mulai dari perusahaan dengan jumlah karyawan kurang dari 100 hingga perusahaan dengan jumlah karyawan lebih dari 10.000 dan mewakili industri pendidikan, energi, hiburan, pemerintah Federal, layanan keuangan, perawatan kesehatan, manufaktur, media, dan teknologi.

Di dalam laporan tersebut, disebutkan bahwa sebanyak 91% responden mengalami insiden keamanan API di tahun 2020. Insiden keamanan API tersebut paling tinggi disebabkan oleh adanya kerentanan (54%) dan masalah autentikasi (46%), diikuti oleh *bot/ scraping* (20%) dan serangan *denial of service* (19%). Adanya Kerentanan ini dapat mengakibatkan *eksfiltrasi* data, penyalahgunaan akun, atau *downtime* layanan.

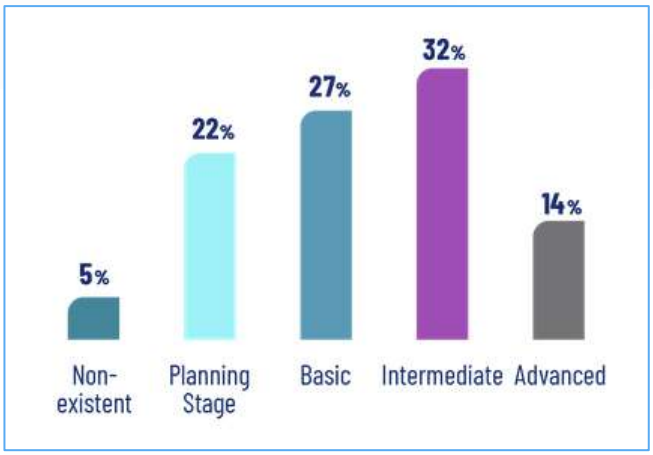
Penyebab terjadinya insiden keamanan pada API



(sumber : https://salt.security/api-security-trends)

Selain itu, Salt Security juga melakukan survei terhadap sejumlah responden untuk mengetahui gambaran strategi keamanan yang digunakan untuk program pengembangan API di perusahaan mereka.

Gambaran strategi keamanan untuk program pengembangan API



(sumber : https://salt.security/api-security-trends)

Berdasarkan data tersebut, sebanyak lebih dari 25% organisasi menjalankan aplikasi berbasis API tanpa strategi keamanan dan 27% organisasi lainnya hanya memiliki strategi dasar untuk keamanan API.

Berdasarkan laporan tersebut, strategi keamanan API menjadi sangat penting untuk diterapkan. Melalui kesadaran keamanan informasi, praktik pengembangan aplikasi yang aman, dan pengujian sistem aplikasi secara menyeluruh, diharapkan dapat membantu meningkatkan keamanan API yang digunakan pada sistem aplikasi.

1. PENDAHULUAN

Microservice merupakan paradigma yang saat ini populer digunakan dalam mengembangkan sistem aplikasi. Hal ini didasari oleh keunggulan *microservice* yaitu *agility*, fleksibilitas, skalabilitas, dan ketersediaan *tools* untuk mengotomatisasi proses-proses dalam *microservice*. Walaupun menawarkan banyak keunggulan, implementasi *microservice* dalam jumlah besar di lingkungan jaringan yang kompleks perlu memperhatikan infrastruktur yang digunakan, baik yang berdiri sendiri maupun satu paket kerangka kerja arsitektur. Infrastruktur penerapan *microservice* tersebut meliputi *Application Programming Interface* (API) *Gateway* dan jejaring layanan (*service mesh*).

Pada dokumen ini akan dijelaskan panduan untuk melakukan analisis implementasi fitur-fitur inti *microservice*, alternatif mode konfigurasi kerangka arsitektur, upaya penanganan ancaman, dan implementasi strategi keamanan yang mengacu pada NIST SP 800-204 *Security Strategies for Microservice-based Application System*.

1.1. Ruang Lingkup

Pembahasan tentang fitur-fitur inti dan kerangka arsitektur yang terdapat dalam dokumen ini terbatas pada masalah-masalah yang berhubungan dengan keamanan implementasi. Adapun fokus utamanya adalah pada metodologi untuk mengembangkan strategi keamanan untuk aplikasi berbasis *microservice* melalui tiga langkah mendasar berikut:

a. Pendalaman tentang teknologi di balik sistem aplikasi berbasis *microservice* yang berfokus pada prinsip-prinsip desain, *basic building blocks*, dan infrastruktur.
b. Tinjauan terkait ancaman khusus terhadap lingkungan operasional *microservice*.
c. Analisis alternatif implementasi terkait dengan fitur inti, alternatif konfigurasi terkait dengan kerangka kerja arsitektur seperti API *gateway* dan *service mesh* serta upaya pencegahan ancaman pada *microservice* dalam rangka mengembangkan strategi keamanan.

1.2. Target Audiensi

Dokumen ini ditujukan kepada:

a. *Chief Security Officer* (CSO) atau *Chief Technology Officer* (CTO) dari departemen TI di perusahaan (swasta) atau lembaga pemerintah yang ingin mengembangkan infrastruktur untuk melakukan *hosting* sistem terdistribusi menggunakan arsitektur *microservice*.

b. Perancang aplikasi yang ingin merancang sistem aplikasi berbasis *microservice*.

1.3. Rujukan Penting Lainnya

Dokumen ini merupakan saduran dari NIST SP 800-204 *Security Strategies for Microservice-based Application System* dan OWASP API *Security* Top 10. Oleh sebab itu, terdapat rujukan lain yang harus diperhatikan dalam mendalami dokumen ini, yaitu:

a. NIST *Special Publication* (SP) 800-190, *Application Container Security Guide.*

b. NIST *Interagency or Internal Report* (NISTIR) 8176, *Security Assurance Requirements for Linux Application Container Deployments.*

1.4. Sistematika Penyusunan Dokumen

Dokumen ini terdiri dari lima bagian dengan detail sebagai berikut:

a. Bagian 1 merupakan pendahuluan yang berisi detail singkat dokumen ini.

b. Bagian 2 memberikan gambaran tentang sistem aplikasi berbasis *microservice*, dimulai dari pandangan konseptual diikuti dengan prinsip-prinsip desain, faktor penggerak bisnis, blok pembangun (*building block*), gaya interaksi komponen, praktik fitur inti, dan kerangka kerja arsitektur.

c. Bagian 3 memberikan gambaran pada tingkat lapisan (teknis *deployment*) dari ancaman pada lingkungan *microservice*.

d. Bagian 4 berisi informasi analisis terkait penerapan fitur inti dalam rangka mendukung aplikasi berbasis *microservice* dan strategi keamanan untuk mengimplementasikan fitur inti berdasarkan analisis implementasinya.

e. Bagian 5 berisi informasi analisis yang berkaitan dengan kerangka arsitektur yang menggabungkan fitur inti dalam infrastruktur aplikasi berbasis *microservice* dan menjabarkan strategi keamanan untuk tiap kerangka arsitektur.

2. LATAR BELAKANG TEKNOLOGI

Bagian ini berisi penjelasan tentang teknologi di balik pengembangan dan penerapan sistem aplikasi berbasis *microservice* dengan menggunakan prinsip desain yang mendasarinya, artefak yang menyusun blok pembangun, dan alternatif konfigurasi untuk menghasilkan skema penerapan *microservice* yang berbeda.

2.1. Konsep Dasar

Sistem aplikasi berbasis *microservice* terdiri dari beberapa komponen (*micro service*) yang saling berkomunikasi satu sama lain melalui panggilan prosedural jarak jauh yang bersifat *synchronous* atau *asynchronous*. Setiap *microservice* biasanya mengimplementasikan satu proses bisnis atau fungsionalitas yang berbeda (misalnya fungsi penyimpanan detail informasi pelanggan, fungsi penyimpanan katalog produk, fungsi pemrosesan pesanan pelanggan, dan lainnya). Setiap *microservice* memiliki logika bisnisnya tersendiri. Beberapa *microservice* akan menyediakan *Representational State Transfer* (REST) API yang dapat digunakan oleh aplikasi klien atau *microservice* lainnya. *Microservice* lainnya yang dimaksud dapat mengimplementasikan *User Interface* (UI) berbasis web. Dalam masa operasi, *microservice* dapat dikonfigurasi untuk dijalankan sebagai proses di server aplikasi, di mesin virtual (VM), atau dalam wadah (kontainer). Aplikasi berbasis *microservice* sering diidentifikasi sebagai aplikasi yang bersifat *cloud-native*.

2.2. Prinsip Desain

Berikut adalah dasar-dasar dari desain *microservice:*

a. Setiap *microservice* harus dikelola, direplikasi, diukur, ditingkatkan, dan digunakan secara independen, terpisah dari *microservice* lainnya.
b. Setiap *microservice* harus memiliki fungsi tunggal dan beroperasi dalam konteks terbatas (yaitu, memiliki tanggung jawab dan tidak bergantung pada *service* lain).
c. Semua *microservice* harus dirancang untuk dapat mengantisipasi kegagalan dan memiliki mekanisme pemulihan yang baik.
d. Dapat memanfaatkan layanan seperti basis data, *cache*, dan direktori untuk manajemen status.

Berdasarkan dasar-dasar tersebut, maka dihasilkan beberapa prinsip sebagai berikut:

a. Otonomi (*Autonomy*), yaitu setiap *service* tidak terlalu bergantung satu sama lain dan proses pengembangan aplikasi dapat berlangsung lebih cepat.
b. *Loose Coupling*, menunjukkan tingkat ketergantungan yang rendah antar *service*, sehingga perubahan pada satu *service* tidak memerlukan perubahan pada *service* lain.
c. Penggunaan kembali (*reusable*), kode yang diimplementasikan dapat digunakan ulang dan mudah dipahami. Hal ini untuk memudahkan proses pengembangan karena kode-kode yang digunakan sudah dipisahkan sesuai dengan jenisnya.
d. Komposit (*Composite*), *service* dapat digunakan pada konteks dan proses bisnis yang berbeda.
e. Toleransi kesalahan (*fault tolerance*), adalah properti yang memungkinkan sistem untuk terus beroperasi dengan baik jika terjadi kegagalan pada beberapa komponennya.
f. Dapat ditemukan (*discoverability*), aplikasi dan *microservices* dapat saling menemukan dalam jaringan.
g. Penyelarasan API dengan proses bisnis.

2.3. Faktor Penggerak Bisnis

Dilihat dari sudut pandang pengguna dan perilaku organisasi, terdapat beberapa faktor yang menjadi penggerak bisnis atau dasar dibutuhkannya aplikasi berbasis *microservice* yaitu:

a. Akses *Ubiquitous* (di mana-mana): pengguna menginginkan akses melalui beberapa perangkat (seperti browser dan perangkat seluler).
b. Skalabilitas: aplikasi harus memiliki tingkat skalabilitas yang baik untuk menjaga ketersediaan dalam menghadapi peningkatan jumlah pengguna dan/atau peningkatan tingkat penggunaan dari basis pengguna yang ada.
c. Mengadopsi metode pengembangan berbasis *agile*, yaitu organisasi menuntut seringnya dilakukan pembaruan untuk merespons perubahan organisasi (proses dan struktural) dan permintaan pasar dengan cepat.

2.4. Blok Pembangun (*Building Block*)

Aplikasi berbasis *microservice* (seperti aplikasi web) dibangun menggunakan gaya arsitektur atau pola desain yang tidak terbatas pada teknologi tertentu dan terdiri dari entitas independen kecil (*endpoint*) yang berkomunikasi satu sama lain menggunakan mekanisme yang ringan *(lightweight)*. *Endpoint* tersebut diimplementasikan menggunakan *Application Programming Interface* (API). Ada beberapa jenis API *endpoint*, seperti *Simple Object Access Protocol* (SOAP) atau Protokol REST *(Representational state transfer)*. Masing-masing *endpoint* menyediakan *service* yang berbeda dan umumnya memiliki penyimpanan atau repositori datanya sendiri. Akses ke *service* ini disediakan oleh berbagai platform atau jenis klien, seperti browser atau perangkat seluler, menggunakan komponen yang disebut "klien". Secara Bersama-sama, komponen *service* dan klien membentuk sistem aplikasi berbasis *microservice*. *Service* dalam sistem tersebut dapat diklasifikasikan sebagai berikut:

a. *Service* fungsionalitas aplikasi.
b. *Service* infrastruktur (disebut "fitur inti" dalam dokumen ini) yang diimplementasikan baik sebagai fitur yang berdiri sendiri atau merupakan gabungan dalam kerangka kerja arsitektur.

Dalam sistem aplikasi berbasis *microservice*, masing-masing *service* yang merupakan bagian kecil dari suatu *service* yang lebih besar dapat dibangun menggunakan teknologi yang berbeda. Hal ini mendukung konsep heterogenitas, yang berarti bahwa setiap *service* dalam sistem aplikasi berbasis *microservice* dapat ditulis dalam bahasa pemrograman yang berbeda, platform pengembangan berbeda, atau menggunakan teknologi penyimpanan data yang berbeda pula.

Service-service pada aplikasi berbasis *microservice* diletakkan pada *node* yang berbeda. Tiap *node* berkomunikasi dari jarak jauh, dan akan menghasilkan latensi yang mempengaruhi kinerja sistem. Oleh karena itu, implementasi *microservice* memerlukan infrastruktur komunikasi yang ringan atau bersifat *lightweight*.

Apabila terdapat *service* yang memiliki kendala kinerja karena tidak cukupnya *Central Processing Unit* (CPU), maka *node* yang menjalankan *service* tersebut dapat ditingkatkan sumber dayanya sesuai kebutuhan. Sementara, *service* lain dapat terus dijalankan menggunakan perangkat keras yang lebih kecil dan lebih murah.

2.5. Gaya Interaksi

Dalam aplikasi yang bersifat monolitik, setiap prosedur atau fungsi akan memanggil fungsi lainnya menggunakan mekanisme panggilan pada tingkat bahasa pemrograman (menggunakan *method* atau *function*). Aplikasi berbasis *microservice* memiliki gaya interaksi yang berbeda, yaitu:

a. Setiap *service* biasanya merupakan proses yang berjalan di *node* jaringannya sendiri melalui mekanisme komunikasi antar-proses atau *interprocess communication* (IPC).

b. Setiap *service* didefinisikan menggunakan *interface definition language* (IDL) (misalnya, Swagger/OpenAPI), sehingga menghasilkan artefak yang disebut API.

c. Langkah pertama dalam pengembangan *service* melibatkan penulisan definisi antarmuka, yang ditinjau oleh pengembang dan klien lalu diulang beberapa kali sebelum *service* diimplementasikan. Dengan demikian, API berfungsi sebagai kontrak antara klien dan *service.*

Pilihan mekanisme IPC menentukan sifat API. Pada Tabel 1 dijelaskan sifat definisi API untuk setiap mekanisme IPC.

Tabel 1. Mekanisme IPC dan Jenis API

Mekanisme IPC	Sifat Definisi API
Asychronous Text-based (misalnya, *Advanced Message Queuing Protocol* (AMQP) atau *Simple Text Oriented Messaging Protocol* (STOMP)).	Terdiri dari saluran pesan dan jenis pesan
Synchronous request/response (misalnya, REST atau *Thrift* berbasis HTTP)	Terdiri dari *Uniform Resource Locators* (URL) dan format *request/response*

Ada berbagai jenis format pesan yang digunakan dalam komunikasi IPC: *text-based* dan *human-readable*, seperti *JavaScript Object Notation* (JSON) atau *Extensible Markup Language* (XML), atau format biner murni yang dapat dibaca mesin, seperti *Apache Avro*.

Prinsip otonomi yang dijelaskan sebelumnya memungkinkan setiap *microservice* menjadi entitas mandiri. Namun, untuk aplikasi berbasis *microservice* yang menyediakan beberapa kemampuan proses bisnis (misalnya, aplikasi belanja *online* yang menyediakan beragam proses bisnis seperti pemesanan, pengiriman,

dan pembuatan faktur), komponen *microservice* selalu bergantung pada *microservice* lain. Sesuai contoh tersebut, fungsi *microservice* 'pengiriman' bergantung pada data dalam *microservice* 'pemesanan' untuk menjalankan fungsinya sehingga menghasilkan *record* pengiriman atau daftar belanja. Oleh karena itu, walaupun *microservice* dapat berdiri sendiri dalam lingkungan otonom, hubungan antara satu *microservice* dengan *microservice* lainnya juga harus tetap dipertahankan. Berbagai pendekatan hubungan antara satu *microservice* dengan *microservice* lainnya sering kali ditentukan oleh proses bisnis dan kebutuhan infrastruktur TI, termasuk pola interaksi, pola pesan, dan mode penggunaan. Dalam dokumen ini, kami menggunakan istilah "pola interaksi". Berikut adalah beberapa contoh pola interaksi:

***Request-Reply*:** Pada saat melakukan *request*, *microservice* membuat *request* khusus atas suatu informasi atau untuk mengambil suatu tindakan. Kemudian, *microservice* tersebut akan menunggu respons. Tujuan dari *request* adalah pengambilan suatu informasi dengan tujuan mempresentasikan data. Pada saat melakukan *reply*, satu *microservice* akan meminta *microservice* lainnya untuk melakukan beberapa tindakan yang melibatkan fungsi bisnis yang dapat mengubah status (misalnya, pelanggan mengubah profil pribadi atau mengirimkan pesanan). Dalam pola *request-reply*, ada ketergantungan selama operasi yang kuat antara dua *microservice* yang terlibat, yang ditunjukkan dalam dua cara berikut:

a. Satu *microservice* dapat menjalankan fungsinya hanya jika *microservice* lainnya tersedia.
b. *Microservice* yang membuat *request* harus memastikan bahwa *request*-nya telah berhasil dikirim ke *microservice* tujuan.

Karena sifat komunikasi dalam protokol ini, maka protokol komunikasi yang bersifat *synchronous* seperti HTTP banyak digunakan. Jika *microservice* diimplementasikan dengan REST API, maka pesan di antara *microservice* menjadi panggilan HTTP REST API. REST API sering didefinisikan menggunakan bahasa standar, seperti RAML (RESTful API *Modeling Language*) yang dikembangkan untuk definisi dan publikasi antarmuka *microservice*. HTTP adalah jenis komunikasi yang dapat melakukan pemblokiran di mana klien yang mengirimkan *request* dapat melanjutkan fungsinya hanya ketika sudah menerima *respons*.

Publish-Subscribe: Pola ini digunakan ketika *microservice* perlu berkolaborasi untuk mewujudkan suatu proses bisnis atau transaksi yang lebih kompleks. Pada lingkungan bisnis, pendekatan ini juga disebut sebagai pendekatan *event-driven* atau pendekatan *event subscription*. Dalam pola ini, *microservice* mendaftarkan dirinya sendiri atau berlangganan terhadap *event*, yang dipublikasikan ke perantara pesan melalui *interface bus*. *Microservice* ini dibangun menggunakan API yang digerakkan oleh *event* dan menggunakan protokol yang bersifat *asynchronous*, seperti *Message Queuing Telemetry Transport* (MQTT), *Advanced Message Queueing Protocol* (AMQP), dan *Kafka Messaging*, yang memiliki fitur notifikasi dan langganan. Dalam protokol yang bersifat *asynchronous*, pengirim pesan biasanya tidak menunggu tanggapan terlebih dahulu, melainkan hanya perlu mengirim pesan ke agen (misalnya, antrean *RabbitMQ*). Salah satu kasus penggunaan untuk pendekatan ini adalah penyebaran pembaruan data ke beberapa *microservice* berdasarkan suatu peristiwa tertentu.

2.6. Praktik Fitur-fitur Inti

Infrastruktur komunikasi merupakan hal yang penting dalam mengimplementasikan berbagai fitur dalam aplikasi berbasis *microservice*. Fitur-fitur ini dapat diimplementasikan baik secara berdiri sendiri atau digabungkan bersama dalam kerangka arsitektur seperti API *gateway* atau *service mesh*. Dalam API *gateway*, fitur-fitur ini dapat diimplementasikan melalui komposisi *service* atau implementasi langsung berbasis kode. Fitur-fitur ini termasuk, tetapi tidak terbatas pada autentikasi dan kendali akses, penemuan layanan (*service discovery*), penyeimbang beban (*load-balancing)*, respons *cache*, pemeriksaan kesehatan aplikasi (*health check*), serta pemantauan (*monitoring*). Adapun deskripsi singkat tentang fitur-fitur ini yaitu:

a. Autentikasi dan kontrol akses: Kebijakan autentikasi dan akses bervariasi sesuai dengan jenis API yang dibuka oleh suatu *microservice*. Beberapa API mungkin merupakan API publik sedangkan yang lainnya mungkin merupakan API privat. Selain itu, mungkin juga beberapa API merupakan API mitra, yang secara khusus hanya tersedia untuk mitra tertentu.
b. *Service discovery*: Dalam sistem terdistribusi (*legacy system*), beberapa *service* dikonfigurasi secara khusus untuk beroperasi di lokasi yang telah ditentukan

(alamat IP dan *port*). Dalam aplikasi berbasis *microservice*, skenario tersebut bisa terjadi dan memerlukan mekanisme *service discovery*, seperti:

1) Terdapat banyak *service* dengan *instances* yang berkaitan dengan *service* dan lokasinya berubah secara dinamis.
2) Setiap *service* diimplementasikan dalam VM, yang mungkin memiliki alamat IP dinamis, terutama ketika di-*hosting* pada layanan *Infrastructure as a service* (IAAS) atau *Software as a service* (SAAS).
3) Adanya fitur *autoscaling* sehingga jumlah sumber daya yang terkait dengan *service* dapat dibuat bervariasi berdasarkan fluktuasi beban.

c. Pemantauan dan analisis keamanan: Bertujuan untuk mendeteksi serangan dan mengidentifikasi faktor degradasi *service* melalui pemantauan *traffic* jaringan dari dan ke *microservice*.

API *gateway* atau *microgateway* umumnya digunakan untuk mengimplementasikan fitur-fitur inti berikut:

d. *Optimized endpoint*, seperti:

1) *Request and response collapsing:* Sebagian besar transaksi bisnis akan melibatkan panggilan ke beberapa *microservice*, sering kali dalam urutan yang telah ditentukan sebelumnya. API *gateway* dapat menyederhanakan situasi secara otomatis dengan membuat beberapa *request* yang diperlukan dan mengembalikan satu respons yang telah digabungkan ke klien.
2) API *Transformation:* API *gateway* dapat menyediakan suatu antarmuka untuk klien, yang berbeda dengan API yang dipanggil untuk memenuhi *request* klien. Fitur ini disebut transformasi API. Transformasi API memungkinkan:
    i. Pengubahan implementasi dan bahkan antarmuka API untuk suatu *microservice*.
    ii. Transisi dari aplikasi monolitik ke aplikasi berbasis *microservice* dengan mengaktifkan akses berkelanjutan ke klien melalui API *gateway* pada saat yang bersamaan secara progresif memisahkan aplikasi monolitik, membuat API *microservice* di latar belakang, dan mengubah konfigurasi transformasi API yang sesuai.

3) *Protocol Translation*: Panggilan dari klien ke suatu *microservice* dapat menggunakan protokol berbasis web, seperti *Hypertext Transfer Protocol Secure* (HTTPS). Sementara, *microservice* yang satu dengan yang lainnya berkomunikasi menggunakan protokol *synchronous*, seperti *Remote Protocol Communication* (RPC), atau protokol *asynchronous*, seperti AMQP. Translasi protokol untuk mengakomodasi *request* klien dapat dilakukan oleh API *gateway*.

e. Pemutus Sirkuit *(Circuit Breaker):* Merupakan fitur untuk menetapkan ambang batas terhadap respons yang gagal ke *microservice*. Apabila nilai ambang batas telah dicapai, maka sistem akan memotong *request* proksi ke sumber daya *service*. Hal ini untuk menghindari kemungkinan kegagalan berjenjang, sehingga terdapat waktu yang cukup untuk menganalisis *log*, melakukan perbaikan yang diperlukan, dan melakukan pembaruan untuk sumber daya yang mengalami kegagalan.

f. *Load Balancing*: Terdapat kebutuhan untuk memiliki beberapa sumber daya dari *service* yang sama. Beban pada sumber daya tersebut harus didistribusikan secara merata untuk menghindari keterlambatan dan kegagalan respons.

g. *Rate Limitting*: Tingkat *request* yang masuk ke *microservice* harus dibatasi untuk memastikan ketersediaan *service* yang berkelanjutan bagi semua klien.

h. *Blue/green Deployment*: Saat versi baru *microservice* diterapkan, *request* dari pelanggan yang menggunakan versi lama dapat dialihkan ke versi baru karena API *gateway* dapat diprogram untuk mengetahui lokasi kedua versi.

i. *Canary Releases*: Setelah suatu versi *microservice* dirilis, akan sangat kecil jumlah *traffic* yang langsung ditujukan ke sumber daya yang baru. Kebanyakan *traffic* masih ditujukan ke sumber daya versi yang lama. Setelah berjalan cukup lama, dan data yang dikumpulkan telah cukup, maka semua *request* ke depannya dapat dialihkan langsung ke versi baru.

2.7. Kerangka Arsitektur

Adapun dua kerangka arsitektur utama yang digunakan dengan menggabungkan fitur-fitur inti sehingga menjamin komunikasi yang andal, tangguh, dan aman dalam aplikasi berbasis *microservice* yaitu:

a. API *gateway*, dengan atau tanpa *microgateway*.
b. *Service mesh.*

Peran kerangka kerja ini dalam lingkungan operasi sistem aplikasi berbasis *microservice* dapat dilihat dalam Tabel 2.

Tabel 2: Peran Kerangka Arsitektur dalam Operasi *Microservice*

Kerangka Arsitektur	Peran Dalam Arsitektur Keseluruhan
API *gateway*	Digunakan untuk mengendalikan *traffic client-service* dan antar *service.*
Service mesh	Digunakan untuk *east-west traffic* murni ketika *microservice* diimplementasikan menggunakan kontainer. Dapat digunakan saat *microservice* ditempatkan di VM atau suatu server aplikasi.

Fungsi khas dalam kedua kerangka arsitektur tersebut meliputi: *service discovery*, penyeimbang beban, deteksi kegagalan, respons kegagalan, dan pemantauan serangan.

2.7.1. API *Gateway*

API *gateway* adalah kerangka arsitektur yang populer untuk sistem aplikasi berbasis *microservice*. Tidak seperti aplikasi monolitik di mana *endpoint* dapat berupa server tunggal, aplikasi berbasis *microservice* terdiri dari beberapa *endpoint*. Oleh karena itu, cukup relevan untuk menyediakan satu pintu masuk bagi semua klien ke beberapa *microservice*. Selain itu, API *gateway* dapat digunakan untuk bertindak sebagai *front-end* bagi *service back-end* ketika dilakukan migrasi dari aplikasi monolitik dengan secara bertahap mengganti komponennya dengan *microservice*.

Fungsi utama API *gateway* adalah untuk selalu mengalihkan *request* masuk ke *service down-stream* yang benar. Selain itu juga berfungsi untuk melakukan translasi protokol dan terkadang juga berfungsi menulis *request*. Dalam beberapa kasus langka, API *gateway* digunakan sebagai bagian dari *Backend for Frontend* (BFF). Semua *request* dari klien terlebih dahulu melalui API *gateway*, yang kemudian akan mengalihkan *request* ke *microservice* yang

sesuai. API *gateway* akan sering menangani *request* dengan memanggil beberapa *microservice* dan menggabungkan hasilnya.

Beberapa API atau *microservice* yang dapat diakses melalui API *gateway* dapat ditetapkan sebagai bagian dari definisi *port input gateway* (misalnya, *mobileAPI* atau *MobileService*) atau ditetapkan secara dinamis melalui operasi penerapan layanan API *gateway* dengan parameter *request* yang berisi nama *service*.

Karena API *gateway* adalah titik masuk untuk *microservice*, maka API *gateway* harus dilengkapi dengan *service* infrastruktur yang diperlukan (selain *service* utama pembentukan *request*), seperti *service discovery*, autentikasi dan kendali akses, *load balancing*, *caching*, penyediaan API khusus untuk setiap jenis klien, pemeriksaan kesehatan (*health check*) aplikasi, pemantauan *service*, deteksi serangan, respons serangan, pencatatan dan pemantauan keamanan, dan pemutus sirkuit. Beberapa fitur tambahan ini dapat diimplementasikan di API *gateway* dengan dua cara, yaitu:

a. Dengan menyusun *service* khusus yang dikembangkan untuk fungsi masing-masing (misalnya, *service registry* untuk *service discovery*).
b. Menerapkan fungsionalitas secara langsung di dalam kode.

Implementasi *gateway*

Untuk mencegah *gateway* mendapati terlalu banyak logika dalam menangani *request* untuk jenis klien yang berbeda, maka perlu adanya pembagian *gateway*. Pola *multiple gateway* ini disebut BFF. Pada BFF, setiap jenis klien diberikan *gateway*-nya sendiri (misalnya, BFF aplikasi web dan BFF aplikasi seluler) sebagai titik pengumpulan untuk *request service*. Masing-masing *backend* harus selaras dengan *front end* yang sesuai (klien) dan biasanya dikembangkan oleh tim yang sama.

Manajemen API untuk aplikasi berbasis *microservice* dapat diimplementasikan melalui arsitektur API *gateway* monolitik atau arsitektur API *gateway* terdistribusi. Dalam arsitektur API *gateway* monolitik, hanya ada satu API *gateway* yang biasanya digunakan di tepi jaringan perusahaan (misalnya, *Demiliterized Zone* (DMZ)) dan menyediakan semua *service* ke API.

Dalam arsitektur API *gateway* terdistribusi, ada beberapa contoh *microgateway*, yang disebarkan lebih dekat ke API *microservice*. *Microgateway* biasanya merupakan API *gateway* yang bersifat *low footprint* dan dapat dibuat dalam bentuk *script*.

Microgateway biasanya diimplementasikan sebagai kontainer yang berdiri sendiri menggunakan *platform* pengembangan seperti *Node.js*. Hal ini berbeda dari proksi pada arsitektur *service mesh* (Bagian 2.7.2), yang diimplementasikan pada *endpoint* API itu sendiri. Ada sejumlah cara agar kebijakan keamanan dapat dimasukkan ke dalam *gateway*. Salah satu pendekatannya adalah dengan mengodekan kebijakan menggunakan format JSON dan memasukkannya melalui antarmuka manajemen kebijakan. *Microgateway* harus berisi kebijakan untuk setiap *request* dan *response* atas aplikasi. Ketika kebijakan dan penegakannya diimplementasikan dalam sebuah kontainer, maka kebijakan tersebut tidak dapat diubah dan dengan demikian memberikan tingkat perlindungan terhadap modifikasi yang tidak disengaja maupun disengaja. Dengan kata lain, jenis modifikasi ini dicegah ketika *microgateway* diimplementasikan sebagai kontainer karena pembaruan kebijakan keamanan sekecil apa pun akan mengharuskan penyebaran ulang *microgateway* tersebut.

2.7.2. *Service Mesh*

Service mesh adalah lapisan infrastruktur khusus yang memfasilitasi komunikasi *service-to-service* melalui *service discovery*, *routing* dan *load balancing* internal, konfigurasi *traffic*, enkripsi, autentikasi dan otorisasi, metrik, dan pemantauan. *Service mesh* memberikan kemampuan untuk secara deklaratif mendefinisikan perilaku jaringan, identitas *node*, dan arus *traffic* melalui kebijakan di lingkungan terhadap perubahan topologi jaringan karena suatu *service* yang datang dan pergi dan berpindah secara berkelanjutan. Hal ini dapat dilihat sebagai model jaringan yang berada di lapisan abstraksi di atas lapisan transport pada model *Open System Interconnection* (OSI) (misalnya, *Transport Control Protocol/Internet Protocol* (TCP/IP)) dan pengalamatan ke *layer session* (*Layer* 5 model OSI). Namun, otorisasi secara terperinci mungkin masih perlu dilakukan di *microservice*

karena hal tersebut adalah satu-satunya entitas yang memiliki pengetahuan penuh tentang logika bisnis. Sebuah *service mesh* secara konseptual memiliki dua modul yaitu *data plane* dan *control plane*. *Data plane* membawa *traffic request* aplikasi antara *instance service* melalui proksi khusus. *Control plane* mengonfigurasi *data plane*, menyediakan titik agregasi untuk telemetri, dan menyediakan API untuk memodifikasi perilaku jaringan melalui berbagai fitur, seperti *load balancing*, pemutusan sirkuit, atau pembatasan kecepatan.

Service mesh membuat suatu server proksi sederhana untuk setiap *service* dalam aplikasi *microservice*. Proksi khusus ini dapat disebut *'sidecar proxy'* dalam bahasa *service mesh*. *Sidecar proxy* membentuk *data plane*, sedangkan operasi yang diperlukan untuk menerapkan keamanan (kontrol akses) diaktifkan dengan menerapkan kebijakan (misalnya, kebijakan kontrol akses) ke dalam *sidecar proxy* dari *control plane*. Hal ini juga memberikan fleksibilitas untuk mengubah kebijakan secara dinamis tanpa mengubah kode *microservice*.

2.8. Perbandingan dengan Arsitektur Monolitik

Untuk membandingkan arsitektur *microservice* dengan arsitektur monolitik, maka perlu untuk membandingkan fitur aplikasi yang dikembangkan, serta contoh aplikasi di kedua arsitektur untuk proses bisnis tertentu. Contoh terperinci perbedaan dari kedua arsitektur tertera pada Lampiran A.

2.9. Perbandingan dengan *Service-Oriented Architecture* (SOA)

Gaya arsitektur *microservice* memiliki banyak kesamaan dengan arsitektur berorientasi layanan (*Service-Oriented Architecture* (SOA)) karena sejumlah konsep teknis berikut:

a. *Service*: Sistem aplikasi menyediakan berbagai fungsi melalui entitas mandiri atau artefak yang disebut *service* yang mungkin memiliki atribut lain seperti terlihat (*visible*) atau dapat ditemukan (*discoverable*), *stateless*, dapat digunakan kembali (*reusable*), dapat disusun (*composable*), atau memiliki keragaman teknologi.

b. Interoperabilitas: Sebuah *service* dapat memanggil *service* lain menggunakan artefak seperti *enterprise service bus* (ESB) untuk SOA atau melalui *remote procedural call* (RPC) melalui jaringan untuk lingkungan *microservice*.
c. *Loose coupling*: ketergantungan antar-*service* yang minim sehingga perubahan dalam satu *service* tidak memerlukan perubahan di *service* lain.

Terlepas dari tiga konsep teknis umum yang dijelaskan di atas, pendapat teknis tentang hubungan antara SOA dan lingkungan *microservice* dapat dijabarkan sebagai:

a. *Microservice* adalah gaya arsitektur yang terpisah.
b. *Microservice* mewakili satu pola SOA.
c. *Microservice* adalah SOA yang disempurnakan.

Pendapat yang paling umum adalah bahwa perbedaan antara SOA dan *microservice* tidak menyangkut gaya arsitektur kecuali dalam realisasi konkretnya, seperti paradigma dan teknologi pengembangan atau penyebaran.

2.10. Keuntungan *Microservice*

a. Untuk aplikasi yang besar, memisahkan aplikasi menjadi komponen-komponen kecil yang kemudian dapat digabungkan dapat menciptakan independensi/kemandirian di antara tim pengembang yang ditugaskan untuk setiap komponen. Setiap tim kemudian dapat mengoptimalkan dengan memilih platform pengembangan, *tools*, bahasa, *middleware*, dan perangkat kerasnya sendiri berdasarkan kesesuaiannya untuk komponen yang sedang dikembangkan.
b. Setiap komponen dapat diskalakan secara independen. Alokasi sumber daya yang ditargetkan menghasilkan pemanfaatan sumber daya secara maksimal.
c. Jika komponen memiliki antarmuka HTTP RESTful, implementasi dapat diubah tanpa mengganggu fungsi keseluruhan aplikasi selama antarmuka tetap sama.
d. Basis kode yang relatif lebih kecil yang terlibat dalam setiap komponen memungkinkan tim pengembang melakukan pembaharuan lebih cepat sehingga aplikasi dapat tersedia dengan cepat untuk merespons perubahan dalam proses bisnis.

e. *Loose Coupling* menjadi komponen-komponen memungkinkan penahanan pemadaman *microservice* sehingga dampaknya terbatas pada layanan tersebut tanpa efek domino pada komponen lain atau bagian lain dari aplikasi.

2.11. Kekurangan *Microservice*

a. Harus ada kegiatan pemantauan untuk semua *microservice* yang ada. Sehingga dibutuhkan adanya infrastruktur yang mendukung yang memiliki kemampuan untuk memantau status setiap *microservice* dan status aplikasi secara keseluruhan.
b. Muncul permasalahan terkait ketersediaan (*availability*), karena setiap komponen *microservice* yang banyak tersebut dapat berhenti berfungsi kapan saja.
c. Harus ada manajemen versi yang mengatur versi yang harus digunakan untuk suatu komponen *microservice*, yang kemungkinan komponen tersebut harus menggunakan versi terbaru untuk beberapa klien, atau menggunakan versi yang lama untuk klien yang lainnya.
d. Harus ada pengujian integrasi, yang dalam pelaksanaannya lebih sulit karena diperlukan lingkungan pengujian tempat semua komponen *microservice* harus bekerja dan berkomunikasi satu sama lain.
e. Harus menerapkan pengelolaan API yang aman di dalam semua proses interaksi yang ada pada aplikasi berbasis *microservice*.
f. Banyak arsitektur web server yang berjalan di DMZ dan dijaga agar tidak diserang, lalu terdapat *service backend* yang digunakan oleh web server, dan basis data yang digunakan oleh *service backend*. *Service backend* dapat bertindak sebagai lapisan yang lebih kuat antara web server dan basis data. Arsitektur *microservice* cenderung melakukan simplifikasi akan hal ini sehingga *service backend* dipisah menjadi *microservice* yang berpotensi lebih terekspos daripada arsitektur monolitik. Hal ini menghasilkan lebih sedikit lapisan perlindungan antara klien dan data sensitif. Oleh karena itu, sangat penting untuk merancang dan mengimplementasikan *microservice* serta model penerapan *service mesh* atau API *gateway* dengan aman.

3. LATAR BELAKANG ANCAMAN

Berikut pendekatan yang digunakan untuk mengulas latar belakang ancaman:

a. Mempertimbangkan semua lapisan yang ada pada aplikasi berbasis *microservice* dan identifikasi potensi ancaman di setiap lapisannya.

b. Mengidentifikasi rangkaian ancaman yang berbeda dan bersifat eksklusif untuk sistem aplikasi berbasis *microservice*.

3.1. Ulasan Sumber Ancaman

Terdapat enam lapisan pada aplikasi berbasis *microservice* yaitu: perangkat keras, virtualisasi, *cloud*, komunikasi, *service*/aplikasi, dan orkestrasi. Setiap lapisan tersebut dapat dianggap sebagai sumber ancaman, dan beberapa masalah keamanan terkait, dijelaskan di bawah ini untuk memberikan gambaran umum tentang latar belakang ancaman pada aplikasi berbasis *microservice*. Penting untuk diingat bahwa banyak dari kemungkinan ancaman tersebut merupakan ancaman umum dan tidak khusus untuk lingkungan aplikasi berbasis *microservice*.

a. Lapisan perangkat keras: Ancaman terhadap perangkat keras cukup jarang terjadi. Dalam konteks dokumen ini, perangkat keras dianggap dapat dipercaya, dan ancaman dari lapisan ini tidak perlu dipertimbangkan.

b. Lapisan virtualisasi: Pada lapisan ini, ancaman terhadap *microservice* atau wadah *hosting* berasal dari *hypervisor* yang disusupi dan penggunaan *image* kontainer dan VM yang berbahaya/rentan.

c. *Cloud environment*: Karena virtualisasi adalah teknologi utama yang digunakan oleh penyedia *cloud*, rangkaian ancaman yang sama terhadap lapisan virtualisasi juga berlaku pada *cloud*. Selanjutnya, ada potensi ancaman dalam infrastruktur jaringan penyedia *cloud*. Misalnya, meng-*hosting* semua *microservice* dalam satu penyedia *cloud* mengakibatkan lebih sedikit kontrol keamanan tingkat jaringan untuk komunikasi antar-proses, dibandingkan dengan kontrol untuk komunikasi antara klien eksternal dengan *microservice* yang di-*hosting* dalam *cloud*.

d. Lapisan komunikasi: Lapisan ini bersifat unik untuk aplikasi berbasis *microservice* karena banyak *microservice* mengadopsi paradigma desain dan gaya interaksi yang berbeda (*synchronous* atau *asynchronous*) di antara setiap *microservice*. Banyak fitur inti dari *microservice* berkaitan dengan lapisan ini, dan

ancaman terhadap fitur inti ini diidentifikasi pada ancaman khusus *microservice* (Bagian 3.2).

e. Lapisan *service*/aplikasi: Pada lapisan ini, ancaman disebabkan oleh kode yang berbahaya. Hal ini termasuk dalam metodologi pengembangan aplikasi yang aman.

f. Lapisan orkestrasi: Lapisan orkestrasi dapat berperan jika implementasi *microservice* melibatkan teknologi seperti kontainer. Ancaman di lapisan ini berkaitan dengan subversi otomatisasi atau fitur konfigurasi, terutama yang terkait dengan penjadwalan dan pengelompokan (*clustering*) server, kontainer, atau VM yang meng-*hosting microservice*.

3.2. Ancaman Khusus Pada *Microservice*

Sebagian besar fitur inti pada penerapan aplikasi berbasis *microservice* mengacu pada lapisan komunikasi. Oleh karena itu, strategi keamanan keseluruhan untuk aplikasi berbasis *microservice* harus melibatkan pemilihan opsi implementasi yang tepat, mengidentifikasi kerangka arsitektur yang mengemas fitur inti tersebut, mengidentifikasi ancaman khusus *microservice*, dan menyediakan cakupan untuk melawan ancaman tersebut dalam opsi implementasi.

Namun, aplikasi berbasis *microservice* juga masih rentan terhadap sebagian besar serangan yang juga menyerang aplikasi web, termasuk serangan *injection*, *encoding and serialization attack*, *cross site scripting* (XSS), *Cross-site Request Forgery* (CSRF), dan HTTP *verb tempering*. Teknik untuk mencegah serangan-serangan tersebut harus dijalankan pula di dalam kode *microservice* sehingga harus dipastikan bahwa pengembang sadar akan hal itu dan tidak berasumsi bahwa API *gateway* atau *service mesh* sudah memberikan semua jaminan keamanan bagi aplikasi *microservice*.

3.2.1. Ancaman Mekanisme *Service Discovery*

Fungsi dasar dalam mekanisme *service discovery* adalah:

a. Registrasi dan deregistrasi *service*.

b. *Service discovery*.

Potensi ancaman keamanan terhadap mekanisme *service discovery* meliputi:

a. Menghubungkan *node* berbahaya ke dalam sistem, melakukan pengalihan komunikasi ke *node* tersebut, dan selanjutnya membahayakan *service discovery*.
b. Adanya basis data *service registry* yang rusak dan mengarah ke pengalihan *request* ke *service* yang salah dan mengakibatkan *denial of service*. Serta pengalihan ke *service* berbahaya yang mengakibatkan seluruh sistem aplikasi *compromise*.

3.2.2. Serangan Berbasis Internet

Meskipun semua aplikasi yang memiliki jaringan atau terdistribusi rentan terhadap serangan berbasis internet, aplikasi berbasis *microservice* lebih rentan terhadap jenis serangan ini karena hal-hal berikut:

a. Tidak seperti aplikasi monolitik yang mengekspos suatu set yang lebih kecil dari antarmuka jarak jauh yang memiliki alamat IP, arsitektur *microservice* hampir selalu mengekspos set yang lebih besar dari antarmuka tersebut. Hal ini disebabkan oleh fakta bahwa aplikasi monolitik mendukung implementasi komponen tunggal dari berbagai fungsi bisnis dan biasanya mengekspos antarmuka yang terkonsolidasi ke semuanya. Sedangkan Aplikasi yang menggunakan arsitektur *microservice* menampilkan banyak komponen yang lebih kecil yang berkoordinasi atau terhubung melalui banyak antarmuka.
b. Aplikasi berbasis *microservice* memiliki peningkatan risiko karena fungsionalitas yang bersifat *exposure*, ketika kontrol keamanan yang diterapkan untuk komponen awal tidak diterapkan, dan langsung mengakses komponen akhir. Kompleksitas sistem yang meningkat secara keseluruhan meningkatkan kemungkinan pengembang dapat menghilangkan pengecekan karena mereka tidak dapat menjelaskan tentang kondisi apa yang dialami klien yang mengakses.

Serangan berbasis internet lainnya adalah serangan *botnet*. Meskipun bukan satu-satunya sarana atau satu-satunya sistem yang terkena serangan *botnet*, kerusakan pada aplikasi berbasis *microservice* dapat

mencakup *stuffing*/penyalahgunaan kredensial, pengambilalihan akun, *page scraping* dan pengambilan data, serta *denial of service*.

3.2.3. Kegagalan berjenjang

Kehadiran beberapa komponen dalam aplikasi berbasis *microservice* meningkatkan kemungkinan kegagalan *service*. Meskipun komponen-komponennya dirancang untuk digabungkan dari sudut pandang penyebaran, ada ketergantungan logis atau fungsional karena banyak transaksi bisnis memerlukan eksekusi beberapa *service* secara berurutan untuk memberikan keluaran yang diperlukan. Oleh karena itu, jika *service* yang berada di hulu dalam logika pemrosesan transaksi bisnis mengalami kegagalan, *service* lain yang bergantung padanya mungkin menjadi tidak responsif/gagal. Fenomena ini dikenal sebagai kegagalan berjenjang.

4. STRATEGI KEAMANAN UNTUK MENERAPKAN FITUR INTI DAN MELAWAN ANCAMAN

Strategi keamanan untuk desain dan penerapan sistem aplikasi berbasis *microservice* mencakup hal-hal berikut:

a. Identitas dan manajemen akses.
b. *Service discovery*.
c. Protokol komunikasi yang aman.
d. *Security monitoring*.
e. Teknik peningkatan ketahanan atau ketersediaan.
f. Teknik peningkatan jaminan integritas.

Melawan ancaman khusus pada aplikasi berbasis *microservice,* seperti:

a. Ancaman terhadap mekanisme *service discovery*.
b. Serangan berbasis internet.
c. Kegagalan berjenjang.

Service discovery merupakan fitur inti dalam *microservice*, dan analisis implementasi harus mempertimbangkan ancaman terhadap mekanisme *service discovery*. Demikian pula, akan dibahas implementasi untuk peningkatan ketahanan atau ketersediaan, hingga membahas tindakan penanganan untuk kegagalan berjenjang.

4.1. Strategi untuk Manajemen Identitas dan Manajemen Akses

Microservice dikemas sebagai API dengan bentuk autentikasi yang melibatkan penggunaan API *key*. Untuk proses otorisasi, diperlukan arsitektur terpusat untuk penyediaan dan penegakan kebijakan akses ke semua *microservice* karena banyaknya *service*, API, dan kebutuhan komposisi *service* untuk mendukung proses bisnis (seperti pemrosesan dan pengiriman pesanan pelanggan). Metode umum komunikasi terkait otorisasi melalui token (seperti JSON web token (JWT), atau berupa token akses OAuth 2.0 yang dikodekan dalam format JSON) juga diperlukan karena masing-masing *microservice* dapat diimplementasikan dalam bahasa pemrograman atau platform yang berbeda. Penyediaan kebijakan dan perhitungan keputusan akses memerlukan suatu server yang ter-otorisasi.

Kerugian dari penerapan kebijakan akses kontrol di setiap *microservice* adalah diperlukan upaya tambahan untuk memastikan bahwa kebijakan lintas sektoral yang berlaku untuk semua API *microservice* diimplementasikan secara seragam.

Setiap perbedaan dalam implementasi kebijakan keamanan antar API memiliki implikasi keamanan untuk seluruh aplikasi berbasis *microservice*, meskipun hanya berlaku untuk kebijakan secara umum, karena kebijakan lebih detail dapat ditentukan di *node* yang lebih dekat dengan *microservice* atau di *microservice* itu sendiri. Selanjutnya, jejak (*footprint*) untuk menerapkan kontrol akses di setiap *node microservice* dapat mengakibatkan masalah kinerja di beberapa *node*. Karena beberapa *node microservice* berkolaborasi untuk melakukan transaksi, masalah kinerja yang terkait dengan *node* dapat dengan cepat mengalir di beberapa *service*. Dengan mempertimbangkan kondisi ini, strategi untuk keamanan identitas dan manajemen akses ke *microservice* diuraikan dalam Strategi 1.

Strategi Keamanan untuk Autentikasi (Strategi 1)

- Autentikasi ke API *microservice* yang memiliki akses ke data sensitif tidak boleh dilakukan hanya dengan menggunakan API *key*. Akses ke API tersebut harus menggunakan token autentikasi yang telah ditandatangani secara digital (misalnya, pemberian kredensial klien) atau diverifikasi dengan suatu sumber yang bersifat otoritatif. Selain itu untuk beberapa *service* mungkin memerlukan token sekali pakai atau *short-lived* token (token yang kedaluwarsa setelah periode waktu yang singkat) untuk membatasi kerusakan yang dapat ditimbulkan oleh token yang diserang.
- Token autentikasi harus diproses (dimana awalnya referensi token dikirim ke RP (*Relying party*)), ditandatangani secara kriptografis, atau dilindungi dengan menggunakan skema HMAC (*Hash-based Method Authentication Code*).
- Setiap API *Key* yang digunakan pada aplikasi harus memiliki batasan (*restriction*) yang ditentukan baik untuk aplikasi seperti *mobile app*, IP *address* dan kumpulan API dimana API *Key* tersebut dapat digunakan.
- Apabila Teknik standar seperti OAuth atau OpenID *connect* diterapkan, Teknik tersebut harus diimplementasikan secara aman.

Strategi Keamanan untuk Manajemen Akses (Strategi 2)

- Kebijakan akses ke semua API dan sumber dayanya harus ditentukan dan diajukan ke *access server*. Kebijakan akses pada level atas (*coarse*) harus ditentukan dan diterapkan di API *gateway*, sementara otorisasi pada level bawah harus ditentukan dan diterapkan lebih dekat ke *node microservice* (*microgateway*) atau terkadang bisa juga di *microservice* itu sendiri.
- Mekanisme *caching*: *microservice* dapat menyimpan data kebijakan ke dalam *cache*; *cache* pada *microservice* ini dapat digunakan ketika *access server* tidak tersedia dan harus dihapus setelah waktu tertentu.
- *Access server* harus memiliki kapabilitas untuk mendukung kebijakan yang terperinci.
- Keputusan akses dari *access server* harus disampaikan kepada individu dan kumpulan *microservice* melalui token yang dikodekan dalam format netral platform (misal token OAuth 2.0 yang dikodekan dalam format JSON).
- Cakupan token otorisasi internal yang ditambahkan oleh *microgateway* atau *decision point* untuk setiap *request* harus dikontrol dengan cermat, misalnya dalam *request* transaksi, token otorisasi internal harus dibatasi cakupannya untuk hanya melibatkan *endpoint* API yang harus diakses untuk transaksi itu.
- API *gateway* dapat dimanfaatkan untuk memusatkan penerapan autentikasi dan *access control* untuk semua *microservice*, mengeliminasi kebutuhan untuk menyediakan autentikasi dan *access control* untuk setiap *service* individu. Jika desain ini dipilih, komponen apa pun yang ditempatkan di jaringan dapat membuat koneksi anonim ke *service* yang melewati API *gateway* dan proteksinya. Kontrol mitigasi seperti autentikasi timbal-balik harus diterapkan untuk mencegah koneksi anonim langsung ke *service*.

4.2. Strategi untuk Mekanisme *Service discovery*

Microservice dapat direplikasi dan ditempatkan di mana saja baik di *data center* atau infrastruktur *cloud* untuk kinerja yang optimal dan alasan penyeimbangan beban. Dengan kata lain, *service* dapat sering ditambahkan atau dihapus dan ditetapkan secara dinamis ke lokasi jaringan mana pun. Oleh karena itu, tidak dapat dihindari dalam arsitektur aplikasi berbasis *microservice* untuk memiliki mekanisme *service discovery*, yang biasanya diimplementasikan menggunakan *service registry*.

Service registry digunakan oleh *microservice* untuk mempublikasikan lokasi mereka dalam proses registrasi dan juga digunakan oleh *microservice* yang ingin menemukan *service* yang terdaftar. Oleh karena itu, *service registry* harus dikonfigurasi dengan pertimbangan kerahasiaan, integritas, dan ketersediaan.

Dalam *service-oriented architectures* (SOA), *service discovery* diimplementasikan sebagai bagian dari *enterprise service bus* (ESB) terpusat. Namun, dalam arsitektur *microservice*, di mana fungsi bisnis dikemas dan disebarkan sebagai *service* di dalam wadah dan berkomunikasi satu sama lain menggunakan panggilan API, penting untuk mengimplementasikan *lightweight message bus* yang dapat mengimplementasikan ketiga gaya interaksi yang disebutkan di Bagian 2.5. Selain itu, cara alternatif agar *service registry* dapat diterapkan, terdiri dari dua dimensi yaitu: cara klien mengakses *service registry,* dan *service registry* terpusat versus *service registry* terdistribusi. Klien dapat mengakses *service registry* menggunakan dua metode utama yaitu: pola *client-side discovery* dan pola *server-side discovery*.

Analisis pola *client-side discovery*

Akses *service registry* pada sisi klien dilakukan dengan membangun kesadaran klien terkait *registry*. Klien melakukan *request service registry* untuk lokasi semua *service* yang diperlukan untuk membuat *request*. Kemudian menghubungi *service* target secara langsung. Meskipun sederhana, opsi implementasi untuk *service discovery* ini memerlukan logika *discovery* untuk diimplementasikan pada setiap bahasa pemrograman dan/atau kerangka kerja yang digunakan untuk implementasi aplikasi klien.

Analisis pola *service-side discovery*

Service-side discovery memiliki dua pola implementasi, yaitu: pola pertama dengan mendelegasikan logika *discovery* ke *service router* khusus yang ditetapkan di lokasi tetap, sementara pola kedua menggunakan server di setiap *microservice* dengan fungsi *Domain Name System* (DNS) *resolver* yang dinamis (yang bekerja dengan server otoritatif *Domain Name System Security Extensions* (DNSSSEC)). Dalam opsi *router* khusus, klien membuat semua *request service* ke *router service* khusus ini, yang kemudian menanyakan *service registry* untuk lokasi *service* yang diminta klien dan meneruskan *request* ke lokasi yang ditemukan. Dalam pola DNS

resolver, setiap *microservice* menyelesaikan penemuan *service*-nya sendiri menggunakan DNS *resolver* bawaannya untuk menanyakan *service registry*. DNS *resolver* menyimpan tabel *service* yang tersedia dan lokasi titik akhirnya (alamat IP). Untuk menjaga agar tabel tetap mutakhir, DNS *resolver* menanyakan *service registry* secara berkala (misal setiap beberapa detik) menggunakan DNS *Service record* (*Service Resource Records* (SRV RRs)) untuk *service discovery*. Karena fungsi *service discovery* melalui DNS *resolver* berjalan sebagai tugas di latar belakang (*background*), *endpoint* (URL) untuk semua *peer microservice* tidak terpengaruh saat adanya *request*.

Strategi yang baik adalah dengan menggunakan kombinasi pola *client-side discovery* dan pola *client-side discovery*. Dengan strategi ini, klien dapat melihat seluruh *service* yang ada berdasarkan daftar yang dimilikinya, dan server *microservice* nantinya tetap menjalankan *service discovery* terhadap *request* dari klien tersebut.

Service Registry **Terpusat versus Terdistribusi**

Dalam implementasi *service registry* terpusat, semua *service* yang ingin mempublikasikan *service* mereka perlu mendaftarkan *service* mereka pada satu titik, dan semua yang mencari *service* tersebut menggunakan satu *registry* untuk menemukannya. Kerugian keamanan dari pola ini adalah terjadinya kegagalan tunggal pada *service registry*, namun konsistensi data lebih terjamin. Dalam *service registry* terdesentralisasi, terdapat beberapa *service registry server*, dan *service* dapat mendaftar pada salah satu server yang ada. Dalam jangka pendek, kerugiannya adalah akan terjadi inkonsistensi data antara berbagai *service registry*. Konsistensi di antara berbagai *service registry server* ini dicapai melalui *broadcast* dari satu server ke server yang lain atau dengan propagasi dari satu *node* ke *node* yang lain melalui data terlampir dalam proses yang disebut *piggybacking*.

Terlepas dari pola apapun yang digunakan untuk *service discovery*, penerapan yang aman dari fungsi *service discovery* harus memenuhi persyaratan konfigurasi *service registry* pada Strategi 3.

Strategi Keamanan untuk Konfigurasi *Service Registry* (Strategi 3)

- *Service registry* harus disediakan melalui server khusus atau merupakan bagian dari arsitektur *service mesh*.
- *Service registry* harus berada dalam jaringan yang telah dikonfigurasi dengan parameter *Quality of Service* (QoS) tertentu untuk memastikan ketersediaan dan ketahanannya.
- Komunikasi antara *service* aplikasi dan *service registry* harus terjadi melalui protokol komunikasi yang aman seperti HTTPS atau *Transport Layer Security* (TLS).
- *Service registry* harus memiliki pemeriksaan validasi untuk memastikan bahwa hanya *service* yang sah yang melakukan registrasi, *refresh operation*, dan *query* basis data untuk melakukan proses *service discovery*.
- Pada *microservice*, proses registrasi dan pencabutan registrasi harus diperhatikan. Saat *service* bermasalah (*down*), atau tidak dapat menangani *request*, maka status registrasi klien yang tertinggal dan tidak dicabut akan menyebabkan masalah terhadap integritas seluruh proses. Oleh karena itu, registrasi/pencabutan registrasi sebaiknya menggunakan pola registrasi pihak ketiga, dan *service* harus dibatasi dalam melakukan *query* ke *service registry*.
- Jika registrasi pihak ketiga diterapkan, proses registrasi/pencabutan registrasi hanya dilakukan setelah proses *health check* pada *service* dilakukan.
- *Service registry* terdistribusi harus diterapkan untuk aplikasi *microservice* berskala besar, dan proses pengelolaan harus dilakukan untuk menjaga konsistensi data di setiap *service registry*.

4.3. Strategi untuk Protokol Komunikasi yang Aman

Komunikasi yang aman antara klien dan *services* (*north-south traffic*) dan antar *services* (*east-west traffic*) sangat penting untuk pengoperasian aplikasi berbasis *microservice*.

Strategi tertentu untuk layanan keamanan seperti autentikasi atau pembentukan koneksi yang aman, dapat ditangani di masing-masing titik *microservice*. Sebagai contoh, dalam *fabric model*, setiap *microservice* memiliki kemampuan untuk berfungsi sebagai klien *Secure Sockets Layer* (SSL) dan server SSL (yaitu, setiap *microservice* adalah *endpoint* SSL/TLS). Dengan demikian, koneksi SSL/TLS dimungkinkan untuk komunikasi antar-*service* atau antar-proses dari

perspektif aplikasi secara keseluruhan. Koneksi ini dapat dibuat secara dinamis (sebelum setiap *request* antar-*service*) atau dibuat sebagai koneksi yang tetap hidup (*keep-alive*). Dalam skema koneksi tetap hidup, "*service* A" membuat koneksi setelah *handshake* SSL/TLS selesai, yaitu proses pertama saat *service* A membuat *request* ke "*service* B." Namun, tidak ada *service* yang menghentikan koneksi meski respons *service* B telah memberikan respons terhadap *request service* A. Koneksi yang sama akan digunakan kembali untuk *request-request* berikutnya. Keuntungan dari skema ini adalah biaya yang muncul akibat SSL/TLS *handshake* dapat dihindari untuk setiap *request*, dan koneksi yang ada dapat digunakan kembali untuk ribuan *request* antar-*service* berikutnya. Dengan demikian, koneksi jaringan antar-*service* akan aman secara permanen untuk semua *request*.

Strategi Keamanan untuk Komunikasi Yang Aman (Strategi 4)

- Klien tidak boleh dikonfigurasi untuk memanggil *service* secara langsung, melainkan diarahkan ke URL *gateway* tunggal.
- Komunikasi klien ke API *gateway* serta komunikasi *service-to-service* harus dilakukan setelah proses autentikasi *mutual* dan terenkripsi (misal, menggunakan protokol *mutual* TLS (mTLS)).
- *Service* yang sering berinteraksi harus menggunakan koneksi TLS yang tetap hidup (*keep-alive*).

4.4. Strategi untuk Pemantauan Keamanan (*Security Monitoring*)

Dibandingkan dengan memantau aplikasi monolitik yang berjalan di suatu server, sistem berbasis *microservice* harus memantau banyak *service*, tiap *service* dapat berjalan di server yang berbeda, atau mungkin di-*hosting* di platform aplikasi yang bermacam-macam. Selain itu, setiap transaksi dalam sistem akan melibatkan setidaknya dua atau lebih *service*.

Strategi Keamanan untuk Pemantauan Keamanan (Strategi 5)

- Pemantauan keamanan harus dilakukan di tingkat *gateway* dan setiap *service* untuk mendeteksi, memperingatkan, dan menanggapi setiap anomali, misalnya serangan yang menggunakan kembali token dan serangan injeksi. Selanjutnya, kesalahan validasi input dan kesalahan parameter tambahan, kerusakan dan *core dumps* harus tercatat dalam log.
- Terdapat dasbor untuk menampilkan status berbagai *service* dan segmen jaringan yang menghubungkannya. Minimal, dasbor harus menampilkan parameter keamanan seperti kegagalan validasi input dan parameter tak terduga yang merupakan tanda dari percobaan serangan injeksi.
- Harus dibuat *baseline* untuk aktivitas atau perilaku normal, perilaku yang krusial terhadap *service*, upaya kontak, dan daftar perilaku lainnya. Penempatan dan kapabilitas *Intrusion Detection System* (IDS) harus diatur sedemikian rupa sehingga penyimpangan dari *baseline* dapat dideteksi.

4.5. Strategi Peningkatan Ketersediaan / Ketahanan

Dalam aplikasi berbasis *microservice*, upaya untuk meningkatkan ketersediaan atau ketahanan suatu *service* diperlukan untuk meningkatkan profil keamanan aplikasi secara keseluruhan. Beberapa teknologi yang umum digunakan meliputi:

- Fungsi pemutus sirkuit.
- Penyeimbangan beban (*Load balancing*).
- Pembatasan laju (*throttling*).

4.5.1. Analisis opsi implementasi Pemutus sirkuit

Strategi untuk mencegah atau meminimalkan kegagalan berjenjang yaitu menggunakan pemutus sirkuit, yang menutup pengiriman data ke komponen (*microservice*) yang gagal melampaui ambang batas tertentu. *Service* yang gagal dengan cepat dibuat keluar dari jaringan, hal ini dapat meminimalkan insiden kegagalan berjenjang sementara log komponen *microservice* yang gagal dianalisis serta diperbaiki dan diperbarui.

Ada tiga opsi untuk penerapan pemutus sirkuit, yaitu: langsung di sisi klien, di sisi server, atau di proksi yang beroperasi antara klien dan server.

Opsi Pemutus Sirkuit Sisi Klien

Dalam opsi ini, setiap klien memiliki pemutus sirkuit terpisah untuk setiap *service* eksternal yang dipanggil. Ketika pemutus sirkuit di klien telah memutus panggilan ke suatu *service*, maka tidak ada pesan yang dikirim ke *service* tersebut, sehingga *traffic* komunikasi di jaringan akan berkurang. Selain itu, pemutus sirkuit tidak perlu diterapkan di *microservice*, sehingga lebih menghemat sumber daya. Namun, menempatkan pemutus sirkuit pada klien memiliki dua kerugian dari sudut pandang keamanan. Pertama, harus diyakinkan pada klien bahwa kode pemutus sirkuit dijalankan dengan benar. Kedua, integritas keseluruhan operasi berisiko karena pengetahuan tentang *service* yang tidak tersedia hanya diketahui secara lokal oleh klien, status tidak tersedia ditentukan berdasarkan status salah satu klien saja, bukan gabungan dari respon *service* yang diterima oleh semua klien.

Opsi Pemutus Sirkuit Sisi Server

Dalam opsi ini, pemutus sirkuit internal pada *microservice* memproses semua *request* klien dan menentukan apakah klien diizinkan untuk menjalankan *service* atau tidak. Keuntungan dari opsi ini adalah pemutusan sirkuit lebih terkontrol karena tidak diletakkan pada masing-masing klien, dan *microservice* memiliki gambaran global tentang frekuensi semua *request* dari klien. *Microservice* juga dapat membatasi *request* dengan mudah, misal untuk meringankan beban server untuk sementara.

Opsi Pemutus Sirkuit di Proksi

Dalam opsi ini, pemutus sirkuit diterapkan dalam proksi, yang terletak di antara klien dan *microservice*, yang menangani semua pesan masuk dan keluar. Dalam hal ini, terdapat dua opsi: satu proksi untuk setiap *microservice* atau satu proksi untuk beberapa *service* (biasanya diterapkan di API *gateway*) yang menyertakan pemutus sirkuit sisi klien dan pemutus sirkuit sisi *service* yang ada di dalam proksi tersebut. Keuntungan dari opsi ini adalah kode klien dan kode *service* tidak perlu dimodifikasi, untuk menghindari masalah kepercayaan dan jaminan integritas. Opsi ini juga memberikan perlindungan tambahan seperti membuat klien lebih tahan terhadap *service* yang salah,

dan melindungi *service* dari kasus di mana satu klien mengirim terlalu banyak *request*, yang mengakibatkan beberapa jenis penolakan *service* ke klien lain yang menggunakan *service* yang sama.

Strategi Keamanan untuk Menerapkan Pemutus Sirkuit (Strategi 6)

Opsi pemutus sirkuit proksi sebaiknya diterapkan untuk membatasi komponen yang terhubung ke proksi. Hal ini menghindari risiko meletakkan kepercayaan pada klien dan *service* (misalnya, menetapkan ambang batas dan memutus *request* berdasarkan ambang batas yang ditetapkan).

4.5.2. Strategi untuk *Load Balancing*

Load balancing (penyeimbangan beban) adalah modul yang tidak terpisahkan di semua aplikasi berbasis *microservice*, tujuan utamanya adalah mendistribusikan beban ke *service*. Untuk menyeimbangkan beban *service*, penyeimbang beban memilih satu *service* dalam *namespace* yang di-*request* menggunakan algoritme antrean (misal *round-robin*) untuk menetapkan antrean *request*.

Strategi Keamanan untuk *Load Balancing* (Strategi 7)

- Program yang mendukung fungsi *load balancing* harus dipisahkan dari pemrosesan *request service*. Sebagai contoh, program yang melakukan *health check* pada server untuk menentukan kumpulan *load balancing* harus berjalan di latar belakang.
- Koneksi jaringan antara penyeimbang beban dan platform *service* harus diperhatikan.
- Jika terdapat DNS *resolver* di depan *microservice,* maka DNS *resolver* harus sinkron dengan *load balancer* untuk menghasilkan satu daftar *microservice* yang di-*request*.

4.5.3. Pembatasan Laju

Sasaran utama pembatasan laju adalah memastikan bahwa *service* tidak kelebihan beban yang memengaruhi ketersediaan *service*. Ketika *request* dari satu klien meningkat, maka *microservice* melanjutkan responsnya ke klien lain. *Microservice* menetapkan batas seberapa sering klien dapat melakukan *request service* dalam jangka waktu tertentu. Ketika batas terlampaui, klien

akan menerima pemberitahuan bahwa limit akses yang diberikan telah terlampaui serta data terkait jumlah limit dan waktu limit *counter* diatur ulang agar klien dapat melanjutkan menerima respon. Sasaran kedua dari pembatasan laju adalah untuk mengurangi dampak dari serangan *Denial of Service* (DoS).

Strategi Keamanan untuk Pembatasan Laju (Strategi 8)

- Kuota atau limit penggunaan *service* atau aplikasi harus didasarkan pada infrastruktur dan persyaratan terkait aplikasi.
- Limit penggunaan *service* harus ditentukan berdasarkan rencana penggunaan API yang ditentukan dengan baik.
- *Microservice* dengan tingkat keamanan tinggi harus menerapkan deteksi terhadap *replay attack*. *Replay attack* merupakan serangan di mana transmisi data yang valid (seharusnya hanya sekali) diulang berkali-kali oleh penyerang yang berhasil memalsukan transaksi yang valid. Deteksi *replay attack* dapat dikonfigurasi untuk mendeteksi sepanjang waktu (100%) atau mendeteksi secara acak.

4.6. Strategi Penjaminan Integritas

Persyaratan jaminan integritas dalam aplikasi berbasis *microservice* terdiri dari dua konteks, yaitu:

- Saat versi baru *microservice* dimasukkan ke dalam sistem.
- Untuk mendukung persistensi (keberlangsungan) sesi selama transaksi.

Pemantauan Terhadap *Service* Baru

Setiap kali versi *microservice* yang lebih baru dirilis, proses implementasinya harus bertahap karena semua klien mungkin tidak siap untuk menggunakan versi baru, dan perilaku versi baru untuk semua skenario dan kasus penggunaan mungkin tidak memenuhi harapan proses bisnis meskipun sudah dilakukan pengujian. Untuk mengatasi situasi ini, dapat digunakan teknik yang disebut dengan *canary release*. Dengan teknik ini, hanya *request* dalam jumlah terbatas yang diarahkan ke versi baru, dan sisanya dialihkan ke versi lama yang sedang beroperasi. Setelah dilakukan pengamatan dalam waktu tertentu dan versi baru dapat menjamin performa dan integritas, maka semua *request* dapat dialihkan ke versi baru.

Strategi Keamanan (Jaminan Integritas) untuk Implementasi *Microservice* Versi Baru (Strategi 9)

- *Traffic* ke versi yang ada (*existing*) dan versi baru harus diarahkan melalui jalur terpusat (misal API *gateway*), untuk memantau proses transisi agar terkendali dan memantau risiko terkait dengan *canary release*. Pemantauan keamanan harus mencakup titik yang menjalankan *service* versi *existing* dan versi baru.
- Lakukan pemantauan terhadap penggunaan versi yang sudah ada (saat ini), bersamaan dengan secara bertahap meningkatkan *traffic* ke versi baru.
- Peningkatan performa dan fungsionalitas dari *service* versi baru menjadi faktor dalam meningkatkan *traffic*-nya.
- Preferensi klien untuk versi *existing* dan baru menjadi pertimbangan saat merancang teknik *canary release*.

Persistensi Sesi

Sangat penting untuk mengirim semua *request* dalam sesi klien ke *microservice* hulu yang sama agar klien dapat menjalankan transaksi lengkap melalui beberapa *request* ke *service* tertentu. Target dari semua *request* harus berada pada *upstream* yang sama dalam sesi tersebut. Kondisi ini disebut persistensi sesi. Situasi yang berpotensi merusak kondisi ini adalah situasi dimana *microservice* menyimpan statusnya secara lokal, dan penyeimbang beban yang menangani *request* individu meneruskan *request* dari sesi pengguna yang sedang berlangsung ke server atau *service* yang berbeda. Salah satu metode untuk menerapkan persistensi sesi adalah dengan *sticky cookie*. Dalam metode ini, terdapat mekanisme untuk menambahkan *cookie* ke respons pertama dari *microservice* hulu ke klien tertentu, berisi informasi server yang memberikan respons. *Request* selanjutnya dari klien akan menyertakan nilai *cookie* tersebut, dan mengarahkan *request* ke *upstream* dan server yang sama.

Strategi Keamanan untuk Menangani Persistensi Sesi (Strategi 10)

- Informasi sesi untuk klien harus disimpan dengan aman.
- Artefak yang digunakan untuk menyampaikan informasi server dalam sesi harus dilindungi.
- Token otorisasi internal tidak boleh diberikan kembali kepada pengguna, dan token sesi pengguna tidak boleh melewati *gateway* untuk digunakan dalam keputusan kebijakan.

4.7. Melawan Serangan Berbasis Internet

Meskipun tidak mungkin untuk melindungi dari semua jenis serangan berbasis Internet termasuk *botnet*, API *microservice* harus dilengkapi dengan kemampuan deteksi dan pencegahan terhadap serangan *credential-stuffing* dan *credential abuse* serta kemampuan untuk mendeteksi *botnet*. Hal ini sangat penting untuk aplikasi yang masing-masing *microservice*-nya dapat dipanggil secara independen dan memiliki kumpulan kredensialnya sendiri. Serangan *credential abuse* dapat dideteksi menggunakan analisis secara *offline* maupun *runtime* (selama operasi). Deteksi serangan *botnet* dapat menggunakan produk *bot manager* atau menggunakan fitur tambahan yang terdapat pada *Web Application Firewall* (WAF).

Strategi Keamanan Mencegah *Credential Abuse* dan *Stuffing Attacks* (Strategi 11)

- Strategi pencegahan selama operasi untuk *credential abuse* lebih diminati daripada strategi *offline*. Perlu ditetapkan ambang batas dari percobaan *login* (misal suatu IP *Address*) yang dilakukan pada waktu tertentu. Jika ambang batas terlampaui, server autentikasi/otorisasi harus melakukan tindakan pencegahan (seperti pemblokiran IP sementara). Fitur ini harus ada terutama saat token digunakan, yaitu untuk mendeteksi penggunaan ulang token.
- Solusi mendeteksi *credential stuffing* yaitu memeriksa dan membandingkan *login* pengguna terhadap basis data kredensial yang dicuri dan memperingatkan pengguna yang sah bahwa kredensial mereka telah dicuri.
- Lakukan konfigurasi IDS dan perangkat deteksi lainnya untuk mendeteksi serangan *denial of service* dan mengeluarkan peringatan sebelum *service* tidak dapat diakses, serta mendeteksi *distributed network probe*.
- Lakukan konfigurasikan pada *host* untuk memindai unggahan *file* dan isi memori serta sistem *file* untuk mengantisipasi ancaman *malware*.

5. STRATEGI KEAMANAN UNTUK KERANGKA ARSITEKTUR *MICROSERVICE*

Terdapat dua kerangka arsitektur yang perlu dipertimbangkan dalam mengimplementasikan strategi keamanan, yaitu *microservice* yang menggunakan API *gateway* atau arsitektur *service mesh*.

Pertimbangan keamanan jika menerapkan API *gateway* adalah memilih platform yang tepat untuk meng-*hosting*-nya, lakukan integrasi dan konfigurasi yang tepat menggunakan kerangka kerja autentikasi dan otorisasi yang andal, lalu lakukan pemantauan dan analisis terhadap *traffic* API *gateway*.

Strategi Keamanan untuk Implementasi API *Gateway* (Strategi 12)

- Integrasikan API dengan aplikasi manajemen identitas untuk membangkitkan kredensial sebelum API diaktifkan.
- API *gateway* yang menjalankan manajemen identitas, harus diintegrasikan dengan *identity providers* (IdPs).
- API *gateway* harus terhubung ke *service* yang dapat menghasilkan token untuk *request* klien (misal menggunakan OAuth 2.0 *Authorization Server*).
- Seluruh informasi *traffic* dipantau dan dianalisis untuk mendeteksi serangan (misal *denial of service*, atau aktivitas berbahaya lainnya) dan mencari penyebab jika terjadi penurunan performa.
- Penerapan *gateway* yang terdistribusi harus memiliki fitur translasi token (*exchange*) antar *gateway*. Token pada *gateway* awal harus memiliki cakupan yang lebih luas dibandingkan dengan token pada *gateway* yang terdekat dengan *microservice* (*microgateways*).

Berbeda dengan penerapan pada API *gateway*, implementasi keamanan pada arsitektur *service mesh* harus mampu memastikan bahwa seluruh parameter dikonfigurasi dengan tepat sesuai dengan kebijakan keamanan yang telah ditentukan, sehingga arsitektur *mesh* yang diterapkan tidak menimbulkan kerentanan baru.

Strategi Keamanan untuk Implementasi *Service Mesh* (Strategi 13)

- Perlu kebijakan untuk menetapkan protokol komunikasi antar-*service* dan menentukan beban *traffic* sesuai dengan persyaratan aplikasi.
- Secara *default* harus dikonfigurasi untuk menerapkan kebijakan akses kontrol untuk semua *service*.
- Hindari konfigurasi yang dapat menyebabkan terjadinya *privilege escalation*.
- Penerapan *service mesh* harus memungkinkan konfigurasi untuk menentukan batas penggunaan sumber daya (misal komputasi dan memori). Tanpa fitur ini, akan muncul potensi terganggunya ketahanan dan ketersediaan aplikasi secara keseluruhan.
- Penerapan *service mesh* harus memungkinkan konfigurasi untuk mengumpulkan dan mengirim data kondisi lingkungan *service*, termasuk data *request*, menuju *service* pemantauan terpusat. Kebijakan pemantauan ini harus tetap dilakukan dengan memperhatikan ketersediaan dan ketahanan *service*. Bila perlu, lakukan implementasi *multi-cluster microservice* untuk menjamin ketersediaan dan ketahanan *service*.
- Untuk aplikasi berbasis *microservice* yang sangat sensitif, segmen jaringan *Layer* 3 harus dikonfigurasi dalam suatu platform orkestrator untuk melengkapi segmen jaringan *Layer* 5 yang dicapai di seluruh lapisan *service mesh*. Hal ini bertujuan untuk mencegah ancaman dari penyerang yang berusaha melewati proksi yang digunakan *service mesh* untuk memblokir *traffic* jaringan.

LAMPIRAN A - PERBEDAAN ANTARA APLIKASI MONOLITIK DAN APLIKASI BERBASIS *MICROSERVICE*

A.1. Perbedaan Desain dan Penerapan

Secara konseptual, arsitektur monolitik pada suatu aplikasi melibatkan pembuatan satu artefak besar yang harus diterapkan secara keseluruhan, sementara aplikasi berbasis *microservice* berisi beberapa fungsi mandiri yang digabungkan secara fleksibel yang disebut dengan *service* atau *microservice*. Tiap *service* pada aplikasi berbasis *microservice* dapat digunakan secara independen. Pada aplikasi monolitik, setiap perubahan pada fungsionalitas tertentu mengharuskan aplikasi untuk dikompilasi ulang, dalam beberapa kasus, harus dilakukan pengujian ulang untuk keseluruhan aplikasi. Sedangkan pada *microservice*, modifikasi pada salah satu *service* tidak berpengaruh terhadap *service* lain, dan hanya perlu melakukan pengujian untuk *service* yang dimodifikasi. Dalam aplikasi monolitik, peningkatan beban kerja karena meningkatnya jumlah pengguna mengharuskan dilakukan peningkatan terhadap sumber daya ke seluruh aplikasi, sedangkan dalam *microservice*, peningkatan sumber daya dapat diterapkan secara selektif ke *service* yang kinerjanya dirasa kurang dari yang diinginkan, sehingga memberikan fleksibilitas dalam upaya skalabilitas.

Beberapa aplikasi monolitik dapat dirancang secara modular, tetapi tidak selalu memiliki modularitas semantik atau logis. Aplikasi yang modular dibangun dari sejumlah komponen dan pustaka yang diambil dari vendor yang berbeda, dan beberapa komponen (seperti basis data) dapat didistribusikan ke seluruh jaringan. Dalam aplikasi monolitik seperti itu, desain dan spesifikasi API dapat mirip dengan arsitektur *microservice*. Namun, perbedaannya adalah API pada *microservice* dapat diakses dari internet sehingga dapat dipanggil dan digunakan kembali secara independen. Perbedaan aplikasi berbasis monolitik dan *microservice* tertera pada Tabel 3.

Tabel 3. Perbedaan Logis antara Aplikasi berbasis Monolitik dan *Microservice*

Aplikasi Monolitik	Aplikasi berbasis *microservice*
Harus diterapkan secara keseluruhan.	Penyebaran *service* secara independen atau selektif.
Perubahan di sebagian kecil aplikasi memerlukan penerapan ulang seluruh aplikasi.	Hanya *service* yang dimodifikasi yang perlu diterapkan ulang.
Skalabilitas melibatkan pengalokasian sumber daya ke aplikasi secara keseluruhan.	Sumber daya di setiap *service* dapat ditingkatkan secara selektif.
Panggilan API bersifat lokal.	API yang dapat diakses internet memungkinkan pemanggilan secara independen dan dilakukan penggunaan ulang.

A.1.1. Contoh Aplikasi untuk Mengilustrasikan Perbedaan Desain dan Penerapan

Sebagai contoh, terdapat Aplikasi Belanja *Online* yang memiliki beberapa modul utama, yaitu:

- Modul untuk menampilkan katalog produk yang ditawarkan oleh penjual dengan gambar produk, nomor produk, nama produk, dan harga satuan.
- Modul untuk memproses pesanan pelanggan dengan mengumpulkan informasi tentang pelanggan (nama, alamat) dan perincian pesanan (nama produk dari katalog, jumlah, harga satuan) serta membuat wadah yang berisi semua item yang dipesan dalam sesi tersebut.
- Modul untuk menyiapkan pengiriman pesanan dan menentukan total pembayaran (total paket yang akan dikirim, jumlah setiap item dalam pesanan, preferensi pengiriman, alamat pengiriman).
- Modul untuk menagih pelanggan dengan fitur melakukan pembayaran menggunakan kartu kredit atau rekening bank.

Pada Tabel 4 ditunjukkan perbedaan desain Aplikasi Belanja *Online* tersebut jika diterapkan berbasis monolitik dan *microservice*.

Tabel 4. Perbedaan Konstruksi Aplikasi antara Aplikasi Berbasis Monolitik dan *Microservice*

Konstruksi Aplikasi	Monolitik	Berbasis *microservice*
Komunikasi antar-modul	Semua komunikasi dalam bentuk prosedur panggilan atau dalam bentuk struktur data internal. Modul yang menangani pemrosesan pesanan membuat panggilan ke modul yang menangani fungsi pengiriman dan menunggu proses sebelumnya selesai (*blocking type synchronous communication*).	Setiap modul dirancang sebagai *service* yang independen. Komunikasi antar-modul menggunakan panggilan API dengan protokol web. *Microservice* pemrosesan pesanan dapat membuat panggilan *request-response* ke *microservice* pengiriman dan menunggu responsnya, atau memasukkan detail pesanan yang akan dikirim dalam antrean pesan untuk diambil secara asinkron oleh *microservice* pengiriman.
Cara menangani perubahan atau *peningkatan* (misalnya suatu modul penagihan pelanggan perlu diubah/ditingkatkan untuk menerima kartu debit)	Seluruh aplikasi harus dikompilasi ulang dan diterapkan kembali setelah melakukan perubahan.	Fungsi penagihan pelanggan dirancang sebagai *microservice* yang terpisah, sehingga *service* dapat dengan mudah dikompilasi ulang dan diterapkan kembali.
Scaling aplikasi, atau penambahan sumber daya aplikasi (misal, modul pemrosesan pesanan perlu ditambahkan CPU dan memori untuk menangani beban yang lebih besar)	Fungsi pemrosesan pesanan melibatkan waktu transaksi yang lebih lama dibandingkan dengan fungsi pengiriman atau pembuatan tagihan pelanggan. Penambahan CPU dan memori pada aplikasi monolitik berarti menambahkan sumber daya ke seluruh aplikasi, meskipun tidak semua fungsi membutuhkan peningkatan.	Cukup lakukan peningkatan untuk perangkat keras *microservice* pemrosesan pesanan. Selain itu, jumlah *microservice* pemrosesan pesanan dapat ditambahkan untuk penyeimbangan beban.

Strategi pengembangan dan penyebaran (*deployment*)	Pengembangan ditangani oleh tim pengembangan dan diuji oleh tim penjaminan mutu (QA), lalu tugas penerapan diberikan kepada tim infrastruktur yang mengawasi alokasi sumber daya yang sesuai untuk penerapan.	Siklus lengkap (dari pengembangan hingga penerapan) ditangani oleh satu tim DevOps untuk setiap *microservice* karena merupakan modul yang relatif kecil dengan fungsionalitas tunggal dan menggunakan platform bawaan (misalnya, OS dan pustaka) yang optimal untuk fungsionalitas tersebut.

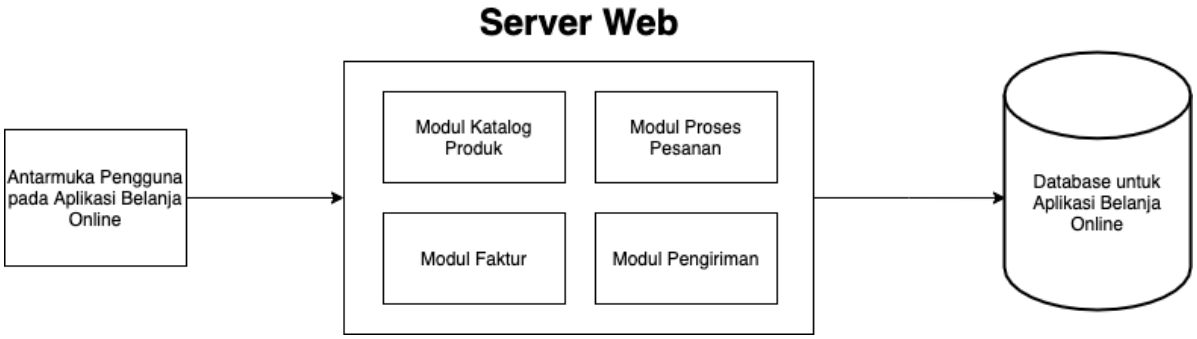
A.2. Perbedaan Selama Operasi

Aplikasi monolitik berjalan sebagai *node* komputasi tunggal sedemikian rupa sehingga *node* menyadari sistem secara keseluruhan atau status aplikasinya. Dalam lingkungan *microservice*, aplikasi dirancang sebagai kumpulan beberapa *node* yang masing-masing menyediakan *service*. Karena mereka beroperasi tanpa perlu berkoordinasi dengan yang lain, status sistem secara keseluruhan tidak diketahui oleh masing-masing *node*. Dengan tidak adanya informasi global atau nilai variabel global, tiap *node* secara individu membuat keputusan berdasarkan informasi yang tersedia secara lokal. Independensi *node* berarti bahwa kegagalan satu *node* tidak mempengaruhi *node* lain. Tidak seperti aplikasi monolitik, di mana *service* dapat berbagi koneksi basis data atau repositori data dan arsitektur *microservice* dapat menyebarkan pola di mana setiap *service* memiliki repositori datanya sendiri. Dalam banyak situasi, interaksi antar-*service* kemungkinan membutuhkan transaksi terdistribusi, yang jika tidak dirancang dengan benar dapat mempengaruhi integritas basis data.

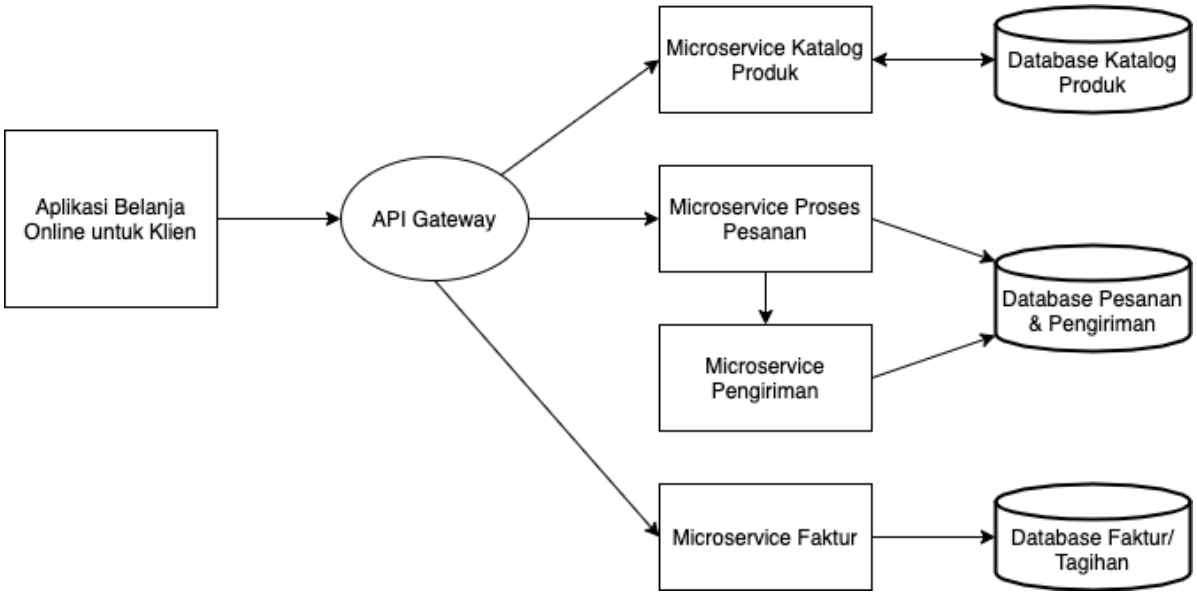
Perbedaan selama operasi antara aplikasi monolitik dan *microservice* beserta implikasinya dirangkum dalam Tabel 5.

Tabel 5. Perbedaan Arsitektur antara Aplikasi Berbasis Monolitik dan *Microservice*

Aplikasi Monolitik	Aplikasi Berbasis *Microservice*
Berjalan sebagai *node* komputasi tunggal, informasi *state* secara keseluruhan sepenuhnya diketahui.	Dirancang sebagai satu kumpulan yang terdiri dari beberapa *node*, tiap *node* menyediakan *service* sendiri, status sistem keseluruhan tidak diketahui oleh tiap *node*.
Dirancang untuk memanfaatkan informasi global atau nilai variabel global.	*Node* individu membuat keputusan berdasarkan informasi yang tersedia secara lokal.
Kegagalan *node* menimbulkan *crash* pada aplikasi.	Kegagalan satu *node* seharusnya tidak mempengaruhi *node* lain.



Gambar 1. Aplikasi Belanja *Online* – Arsitektur Monolitik



Gambar 2: Aplikasi Belanja *Online* – Arsitektur *Microservice*

LAMPIRAN B - PENELUSURAN STRATEGI KEAMANAN FITUR ARSITEKTUR *MICROSERVICE*

Semua strategi keamanan yang dibahas pada bagian 4 & 5 (dengan jumlah total 13 strategi) tercantum dalam Tabel 6 bersama dengan fitur inti *microservice* atau kerangka arsitektur yang terkait dengan masing-masing strategi tersebut.

Tabel 6. Strategi Keamanan pada *Microservice*

Strategi	Strategi Keamanan	Fitur Inti *Microservice*/ Kerangka Arsitektur
Strategi 1	• Autentikasi ke *microservice* API yang memiliki akses ke data sensitif tidak boleh dilakukan hanya dengan menggunakan API *key*. Akses ke API semacam itu memerlukan token autentikasi yang telah ditandatangani secara digital (misalnya, pemberian kredensial klien) atau yang diverifikasi dengan sumber resmi. Selain itu, beberapa *service* mungkin memerlukan token sekali pakai atau token berumur pendek (token akan kadaluwarsa setelah periode waktu telah selesai) untuk membatasi kerusakan yang dapat disebabkan oleh token yang disusupi. • Token autentikasi harus berbasis pegangan, ditandatangani secara kriptografis, atau dilindungi oleh skema HMAC. • Setiap API *key* yang digunakan dalam aplikasi harus memiliki batasan yang ditentukan baik untuk aplikasi (misalnya, aplikasi seluler, alamat IP) dan kumpulan API pada tempat aplikasi tersebut dapat digunakan. • Cakupan pembatasan untuk fungsionalitas setiap API *key* harus sepadan dengan tingkat jaminan yang diberikan selama pemeriksaan identitas, baik pemeriksaan identitas yang digerakkan oleh mesin atau manusia. • Ketika token autentikasi *stateless* (misalnya, JSON Web *Tokens* (JWT)) digunakan dengan mengimplementasikan pustaka bersama yang terkait dengan *microservice*, tindakan	Autentikasi

	pencegahan keamanan berikut harus diperhatikan: (a) waktu kedaluwarsa token harus sesingkat mungkin karena menentukan durasi sesi dan sesi aktif tidak dapat dicabut, dan (b) kunci rahasia token tidak boleh menjadi bagian dari kode pustaka, dan harus berupa variabel dinamis yang diwakili oleh variabel lingkungan atau ditentukan dalam *file* data lingkungan. Nilai kunci harus disimpan dalam brankas data sebagai solusinya. • Jika teknik berbasis standar seperti OAuth atau OpenID *connect* diimplementasikan, teknik tersebut harus digunakan dengan aman.	
Strategi 2	• Kebijakan akses ke semua API dan sumber dayanya harus ditentukan dan disediakan ke akses server. Kebijakan akses pada tingkat *coarse* (level atas) menyatakan "Izin Pemanggilan untuk serangkaian fungsi ditentukan alamatnya" harus ditentukan dan diterapkan di API *gateway* awal sementara otorisasi pada tingkat *granularity* (level bawah) yang lebih kecil (misalnya, terkait dengan domain logika bisnis *microservice* tertentu) harus ditentukan dan diterapkan lebih dekat ke lokasi *microservice* (misalnya, di *microgateway*) atau terkadang di *microservice* itu sendiri. • Mekanisme *Caching*: Mungkin tepat untuk mengizinkan *microservice* untuk menyimpan data kebijakan dalam *cache*; *cache* ini hanya dapat diandalkan ketika server akses tidak tersedia dan akan kedaluwarsa setelah durasi yang sesuai untuk lingkungan/ infrastruktur. • Server akses harus mampu mendukung kebijakan yang lebih mendetail (level bawah). • Keputusan akses dari server akses harus disampaikan kepada individu dan kumpulan *microservice* melalui token standar yang dikodekan dalam format platform-netral (misalnya, token OAuth 2.0 yang dikodekan dalam format JSON). Token dapat berupa token berbasis pegangan atau token bantalan pernyataan. • Cakupan token otorisasi internal yang ditambahkan oleh *microgateway* atau titik keputusan untuk setiap *request* harus	Manajemen akses

	dikontrol dengan hati-hati; misalnya, dalam *request* transaksi, otorisasi token internal harus dibatasi cakupannya hanya melibatkan *endpoints* API yang harus diakses untuk transaksi tersebut. • API *gateway* dapat dimanfaatkan untuk memusatkan penegakan autentikasi dan mengontrol akses untuk semua *downstream microservice*, menghilangkan kebutuhan untuk menyediakan autentikasi dan kontrol akses pada setiap *service* individual. Jika desain ini dipilih, komponen apa pun yang diposisikan dengan tepat di jaringan dapat membuat koneksi anonim ke *service* yang melewati API *gateway* dan perlindungannya. Kontrol mitigasi seperti autentikasi timbal balik harus dimanfaatkan untuk mencegah koneksi anonim langsung ke *service*."	
Strategi 3	• Kemampuan *service registry* harus disediakan melalui server yang didedikasikan atau bagian dari arsitektur *service mesh*. • *Service registry* harus berada dalam jaringan yang telah dikonfigurasi dengan parameter QoS tertentu untuk memastikan ketersediaan dan ketahanannya. • Komunikasi antara *service* aplikasi dan *service registry* harus melalui protokol komunikasi yang aman, seperti HTTPS/TLS. • *Service registry* harus memiliki pemeriksaan validasi untuk memastikan bahwa hanya *service* yang sah yang melakukan operasi pendaftaran dan penyegaran atau menanyakan basis datanya untuk menemukan *service*. • Konteks terikat dan prinsip *loose coupling* untuk *microservice* harus diperhatikan untuk fungsi pendaftaran/pencabutan pendaftaran *service*; *service* aplikasi tidak boleh memiliki hubungan yang erat dengan *service* infrastruktur, seperti *service registry*, dan pola *service self-registration/deregistration* harus dihindari. Selain itu, ketika *service* aplikasi mogok atau sedang berjalan tetapi tidak dalam posisi untuk menangani *request*, *service* aplikasi tersebut tidak dapat melakukan deregistrasi, sehingga mempengaruhi integritas seluruh proses. Pendaftaran atau pembatalan pendaftaran *service* aplikasi harus diaktifkan menggunakan pola pendaftaran	Konfigurasi pendaftaran *service*

	pihak ketiga, dan *service* aplikasi harus dibatasi hanya untuk menanyakan *service registry* untuk informasi lokasi *service* seperti yang dijelaskan dalam pola penemuan sisi klien. • Jika pola pendaftaran pihak ketiga diterapkan, pendaftaran/pencabutan pendaftaran hanya boleh dilakukan setelah melakukan *health check* pada *service* aplikasi. • *Service registry* terdistribusi harus diterapkan untuk aplikasi *microservice* besar, dan harus berhati-hati untuk menjaga konsistensi data di antara beberapa *service registry*.	
Strategi 4	• Klien tidak boleh dikonfigurasi untuk memanggil target *service*-nya secara langsung melainkan dikonfigurasi untuk menunjuk ke URL *gateway* tunggal. • Komunikasi klien ke API *gateway* serta *Service-to-service* harus dilakukan setelah autentikasi bersama dan dienkripsi (misalnya, menggunakan protokol mTLS). • *Service* yang sering berinteraksi harus membuat koneksi TLS tetap aktif.	Komunikasi yang aman
Strategi 5	• Pemantauan keamanan harus dilakukan pada tingkat *gateway* dan *service* untuk mendeteksi, memperingatkan, dan menanggapi perilaku yang tidak pantas, misalnya serangan penggunaan kembali token pembawa dan serangan injeksi. Selanjutnya, kesalahan validasi input dan kesalahan parameter tambahan, *crash* dan *core dump* harus dicatat. Kelas perangkat lunak yang dapat melakukannya adalah OWASP AppSensor yang berpotensi diimplementasikan di *gateway*, *service mesh*, dan *microservice* itu sendiri. • Dasbor pusat menampilkan status berbagai *service* dan segmen jaringan yang menghubungkannya. Minimal, dasbor harus menunjukkan parameter keamanan seperti kegagalan validasi input dan parameter tak terduga yang merupakan tanda nyata dari upaya serangan injeksi. • Sebuah *baseline* untuk perilaku normal yang tidak berada dalam bahaya dalam hal hasil keputusan logika bisnis, upaya kontak, dan perilaku lainnya harus dibuat. Penempatan dan kemampuan *node* IDS harus sedemikian rupa sehingga penyimpangan dari *baseline* ini dapat dideteksi.	Pemantauan Keamanan

Strategi 6	• Opsi pemutus sirkuit proksi harus diterapkan untuk membatasi komponen terpercaya menjadi proksi, yang menghindari kebutuhan untuk menempatkan kepercayaan pada klien dan *microservice* (misalnya, menetapkan ambang batas dan memutus *request* berdasarkan ambang batas yang ditetapkan) karena terdiri dari banyak komponen.	Penerapan Pemutus sirkuit
Strategi 7	• Fungsi penyeimbangan beban harus dipisahkan dari *request service* individual; misalnya, program yang melakukan pemeriksaan kesehatan pada *service* untuk menentukan kumpulan penyeimbang beban harus berjalan secara asinkron di latar belakang. • Perawatan harus dilakukan untuk melindungi koneksi jaringan antara penyeimbang beban dan platform *microservice*. • Ketika *resolver* DNS dikerahkan di depan *microservice* sumber untuk menyediakan tabel *microservice* target yang tersedia, *resolver* tersebut harus bekerja bersama dengan program pemeriksaan kesehatan untuk menyajikan satu daftar ke *microservice* pemanggil.	Penerapan *Load Balancing*
Strategi 8	• Kuota atau limit penggunaan aplikasi harus didasarkan pada infrastruktur pembatasan laju dan persyaratan terkait aplikasi. • Limit harus ditentukan berdasarkan rencana penggunaan API yang ditentukan dengan baik. • Untuk *microservice* dengan tingkat keamanan tinggi, deteksi *replay attack* harus diterapkan. Berdasarkan risikonya, fitur ini dapat dikonfigurasi untuk mendeteksi *replay* sepanjang waktu (100%) atau melakukan deteksi acak.	Pembatasan Laju

Strategi 9	• *Traffic* ke versi yang ada dan versi baru dari *service* harus dialihkan melalui *node* pusat, seperti API *gateway*, untuk memantau bahwa transisi biru/hijau terjadi secara terkendali dan untuk memantau risiko yang terkait dengan *canary release*. Pemantauan keamanan harus mencakup *node* yang meng-*hosting* versi yang ada dan yang lebih baru • Pemantauan penggunaan versi yang ada harus mendorong tingkat "peningkatan" *traffic* ke versi baru. • Kinerja dan kebenaran fungsional versi baru harus menjadi faktor dalam peningkatan *traffic* ke versi baru. • Preferensi klien untuk versi (yang sudah ada atau yang baru) harus dipertimbangkan saat merancang teknik *canary release*.	Induksi versi baru dari *microservice*
Strategi 10	• Informasi sesi untuk klien harus disimpan dengan aman • Artefak yang digunakan untuk menyampaikan informasi server yang mengikat harus dilindungi • Token otorisasi internal tidak boleh diberikan kembali kepada pengguna, dan token sesi pengguna tidak boleh melewati gerbang untuk digunakan dalam keputusan kebijakan.	Penanganan Persistensi Sesi
Strategi 11	• Strategi pencegahan selama operasi untuk penyalahgunaan kredensial lebih disukai daripada strategi *offline*. Ambang batas untuk interval waktu yang ditentukan dari lokasi tertentu (misalnya, alamat IP) untuk jumlah upaya *login* harus ditetapkan; jika ambang batas terlampaui, tindakan pencegahan harus dipicu oleh server autentikasi/otorisasi. Fitur ini harus ada saat token pembawa digunakan, untuk mendeteksi penggunaannya kembali dan menerapkan pencegahan. • Solusi deteksi *credential stuffing* memiliki kemampuan untuk memeriksa *login* pengguna terhadap basis data kredensial yang dicuri dan memperingatkan pengguna yang sah bahwa kredensial mereka telah dicuri. • Konfigurasikan IDS dan perangkat batas untuk mendeteksi hal berikut: (a) serangan *denial of service* dan tingkatkan peringatan sebelum *service* tidak lagi dapat diakses, dan (b) pemeriksaan jaringan terdistribusi.	Pencegahan *credential abuse* dan *stuffing attack*

	• Konfigurasikan *host service* untuk memindai unggahan *file* dan konten memori dan sistem *file* setiap penampung untuk mendeteksi ancaman *malware* yang ada.	
Strategi 12	• Integrasikan API *gateway* dengan aplikasi manajemen identitas untuk menyediakan kredensial sebelum mengaktifkan API. • Saat manajemen identitas dipanggil melalui API *gateway*, konektor harus disediakan untuk diintegrasikan dengan penyedia identitas (IdP). • API *gateway* harus memiliki konektor ke artefak yang dapat menghasilkan token akses untuk *request* klien (misalnya, Server Otorisasi OAuth 2.0). • Salurkan semua informasi *traffic* dengan aman ke aplikasi pemantauan dan/atau analitik untuk mendeteksi serangan (misalnya, penolakan *service*, tindakan jahat) dan menemukan penjelasan atas penurunan kinerja. • Penyebaran *gateway* terdistribusi (atau kombinasi *gateway* awal (yang melakukan intersepsi terhadap semua akses klien) dan *microgateway* (lebih dekat ke *microservice*)) harus memiliki *service* translasi token (*exchange*) antar-*gateway*. Token yang disajikan ke *gateway* awal harus memiliki izin dengan cakupan yang luas sedangkan token yang disajikan ke dalam *gateway* (atau *microgateway*) harus memiliki cakupan yang lebih sempit dengan izin khusus atau jenis token yang sama sekali berbeda yang sesuai untuk platform *microservice* target. Ini membantu untuk menerapkan paradigma hak istimewa paling rendah.	Konfigurasi API *gateway*
Strategi 13	• Dukungan kebijakan harus diaktifkan untuk: (a) menetapkan protokol komunikasi khusus antara pasangan *service* dan (b) menentukan beban *traffic* antara pasangan *service* berdasarkan persyaratan aplikasi • Konfigurasi *default* harus selalu mengaktifkan kebijakan kontrol akses untuk semua *service* • Hindari konfigurasi yang dapat menyebabkan eskalasi hak istimewa (misalnya, izin peran *service* dan pengikatan peran *service* ke akun pengguna *service*)	Konfigurasi *service mesh*

	• Penerapan *service mesh* harus memiliki kemampuan konfigurasi untuk menentukan batas penggunaan sumber daya untuk komponennya. Ketiadaan fitur ini membuat komponen tersebut berpotensi untuk memengaruhi ketahanan dan ketersediaan aplikasi *microservice* secara keseluruhan. • Penerapan *service mesh* harus memiliki kemampuan konfigurasi untuk mengumpulkan dan mengirim metrik lingkungan, termasuk metrik *request*, ke *service* terpusat untuk pemantauan. Kebijakan harus memungkinkan penentuan *service mesh* tunggal atau beberapa *service mesh* (masing-masing dengan bidang kontrolnya sendiri) untuk lingkungan *microservice multicluster* guna memastikan ketersediaan dan ketahanan yang tinggi dalam skenario tersebut. • Untuk aplikasi berbasis *microservice* yang sangat sensitif, segmentasi jaringan *Layer* 3 harus dikonfigurasi dalam platform pengorkestra untuk melengkapi segmentasi jaringan *Layer* 5 yang dicapai di seluruh lapisan *service mesh*. Ini adalah tindakan penanggulanan terhadap ancaman oleh aktor jahat yang menghindari atau melewati *sidecar proxy* yang digunakan *service mesh* untuk melindungi dengan *firewall* dan memblokir *traffic* jaringan.	

LAMPIRAN C - OWASP API *SECURITY* TOP 10 (2019)

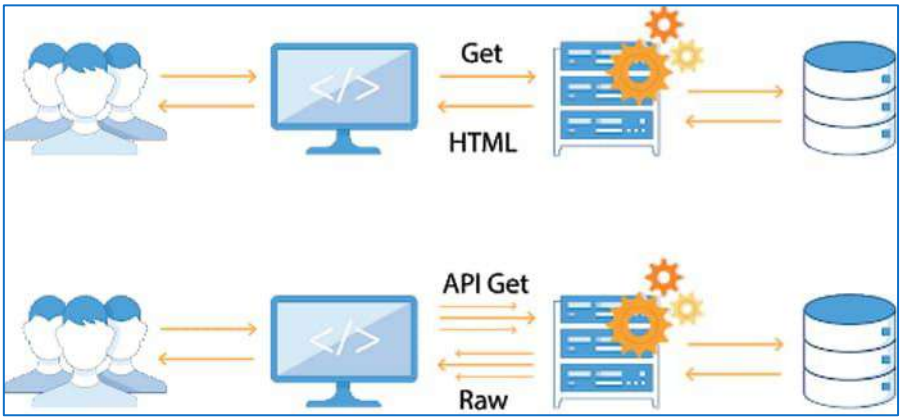
Open Web Application Security Project (OWASP) merupakan komunitas *online* non-profit dibalik terbitnya OWASP Top 10. OWASP menghasilkan artikel, metodologi, dokumentasi, *tools*, dan teknologi untuk meningkatkan keamanan aplikasi.

Sejak tahun 2003, proyek OWASP Top 10 telah menjadi rujukan bagi kerentanan aplikasi web serta cara-cara untuk menanggulanginya. Namun, munculnya API telah mengubah pandangan keamanan secara mendasar sehingga diperlukan pendekatan baru. Pada tahun 2019, OWASP membuat versi Top 10 khusus untuk keamanan API. OWASP API *Security* Top 10 pertama dirilis pada tanggal 31 Desember 2019.

Keamanan Aplikasi Web vs Keamanan API

Pada aplikasi web konvensional, pemrosesan data dilakukan di sisi server, dan halaman web yang dihasilkan kemudian dikirim ke browser klien untuk dirender. Hal ini menyebabkan titik masuk ke arsitektur jaringan relatif sedikit dan mudah untuk dilindungi, misal dengan menggunakan *Web Application Firewall* (WAF) di depan server aplikasi.

Berbeda dengan aplikasi berbasis API, *user interface* menggunakan API untuk mengirim dan menerima data dari server untuk menjalankan fungsi aplikasi. Hal ini menyebabkan klienlah yang melakukan *rendering* dan mempertahankan status (*maintain states*).



Gambar 3. Perbedaan *traffic* aplikasi web konvensional dan API[1]

Ditambah lagi dengan arsitektur *microservice* dimana komponen aplikasi individual dijalankan menggunakan API, hal ini meningkatkan titik serangan secara signifikan.

[1] https://apisecurity.io/encyclopedia/content/owasp/owasp-api-security-top-10.htm

Akibatnya, titik masuk ke arsitektur jaringan sama dengan jumlah API yang memanggil ke *server backend*.

Dengan begitu banyak titik masuk yang tersebar luas dalam arsitektur jaringan, menempatkan *firewall* di depan server tidak lagi cukup untuk memastikan semua titik masuk terlindungi. Selain itu, solusi berbasis WAF tradisional tidak dapat membedakan peretasan *request* API pada *traffic* API yang sah.

Secara alami, API mengekspos logika aplikasi dan data sensitif sehingga menjadi target empuk penyerang. API juga menyediakan fitur kontrak, akan tetapi tidak memuat langkah-langkah untuk memastikan bahwa kontrak ditegakkan, menimbulkan risiko keamanan yang serius pada *service backend* yang terhubung dengan API.

Oleh karena itu, keamanan API memerlukan proyek keamanannya sendiri yang berfokus pada strategi dan solusi untuk memahami dan mengurangi kerentanan dan risiko keamanan API.

OWASP API *Security* Top 10

Perubahan paradigma yang disebutkan di atas telah mengakibatkan OWASP meluncurkan proyek terpisah yang didedikasikan pada keamanan API. Saat ini sudah ada OWASP API *Security* Top 10 yang berfokus secara khusus pada sepuluh kerentanan teratas dalam keamanan API.

OWASP API *Security* Top 10 mencakup dua hal, yaitu:

1. Peran penting API dalam arsitektur aplikasi saat ini dan juga dalam keamanan aplikasi.
2. Munculnya masalah khusus API yang perlu menjadi atensi dan perhatian.

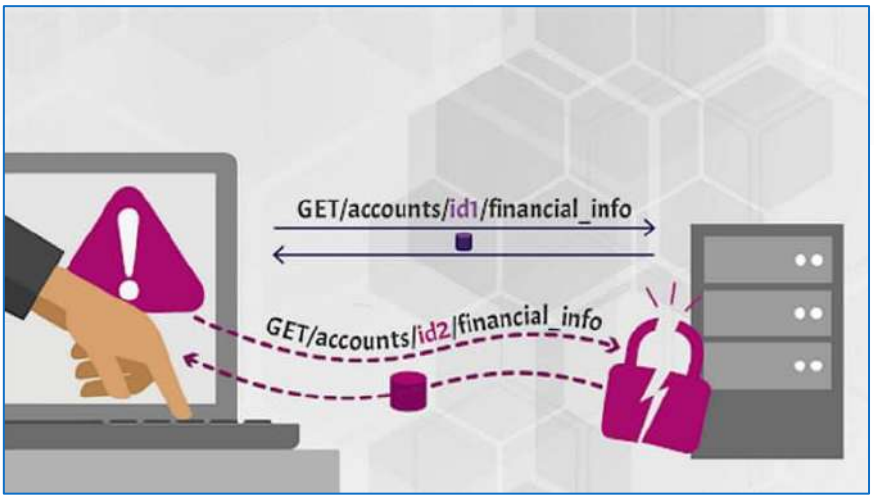
API 1 Otorisasi Tingkat Objek yang Rusak

Spesifik API	Dapat Dieksploitasi: 3	Prevalensi: 3	Dapat dideteksi: 2	Teknis: 3	Spesifik Bisnis
Penyerang dapat mengeksploitasi API *server* yang rentan terhadap otorisasi tingkat objek yang rusak dengan memanipulasi ID dari sebuah objek yang dikirim bersamaan dengan *request* API. Hal ini dapat mengarah pada akses yang tidak sah terhadap data sensitif. Kerentanan ini cukup umum pada aplikasi berbasis API karena server biasanya tidak sepenuhnya melacak status klien, dan lebih mengandalkan pada parameter berupa objek ID, yang dikirimkan dari klien untuk menentukan objek mana yang akan diakses.		Hal ini cukup umum dan sangat berdampak pada serangan API. Otorisasi dan mekanisme kontrol akses pada aplikasi modern sangat rumit dan tersebar luas. Para pengembang cenderung melupakan pemeriksaan infrastruktur sebelum mengakses objek yang sensitif.		Akses yang tidak sah dapat membocorkan data kepada pihak yang tidak berwenang, hilangnya data atau terjadi manipulasi data, hingga mengambil alih akun/sistem secara penuh.	

Detail Kerentanan

Otorisasi tingkat objek adalah mekanisme kontrol akses yang biasanya diterapkan pada tingkat kode untuk memvalidasi bahwa satu pengguna hanya dimungkinkan mengakses objek yang dapat diaksesnya.

Setiap server API yang menerima objek ID, dan melakukan segala jenis tindakan pada objek, harus menerapkan pengecekan otorisasi tingkat objek. Pengecekan dilakukan dengan memvalidasi bahwa pengguna yang terhubung memang memiliki akses untuk melakukan *request* pada objek yang diminta. Kegagalan dalam mekanisme ini menyebabkan kebocoran, modifikasi atau penghancuran data.



Contoh Skenario Serangan

Skenario 1

Suatu platform *e-commerce* menyediakan halaman diagram grafik penjualan untuk toko yang terdaftar. Dengan menginspeksi *request* browser, penyerang dapat mengidentifikasi *endpoint* API yang digunakan sebagai sumber data untuk diagram tersebut, yaitu dengan pola **/shops/{ShopName}/revenue_data.json**. Misal penyerang menggunakan *endpoint* API yang lain dan berhasil mendapatkan daftar semua nama toko yang terdaftar. Dengan skrip yang sederhana untuk memanipulasi nama toko, yaitu mengganti **{shopName}** pada URL, penyerang mendapatkan akses ke data penjualan dari ribuan toko pada *e-commerce* tersebut.

Skenario 2

Misal penyerang melakukan pemantauan terhadap *traffic* jaringan perangkat *wearable*. Suatu *request* HTTP PATCH berikut menarik perhatian penyerang karena adanya *header request* HTTP khusus **X-User-Id: 54796** (diasumsikan sebagai nomor pengguna dari perangkat). Dengan mengganti nilai **X-User-Id** menjadi **54795,** penyerang berhasil menerima respon dari HTTP dan dapat mengubah akun data pengguna lain.

Cara Mencegah

- Menerapkan mekanisme otorisasi yang tepat yang bergantung pada kebijakan pengguna serta hierarkinya.

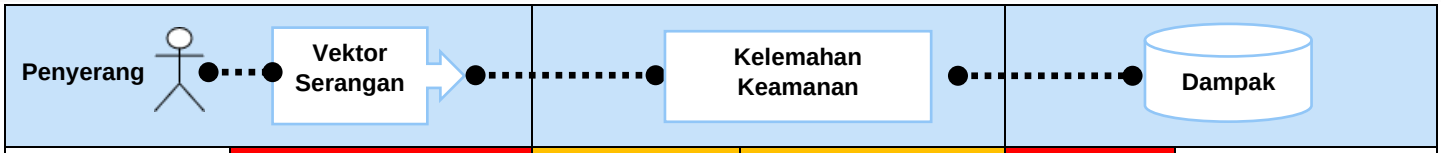
- Gunakan mekanisme otorisasi untuk memeriksa apakah pengguna yang masuk memiliki akses dalam melakukan *request* pada setiap fungsi yang membutuhkan input dari klien dalam mengakses *record* di basis data.
- Menggunakan nilai acak untuk IDs.
- Jangan mengandalkan ID yang dikirim oleh klien, sebaiknya gunakan ID yang disimpan di dalam sesi.

Contoh Kasus Implementasi

```
if($this->session->userdata('role')!="admin"){
    $message=[
        'result'=>"gagal"
    ];

    $rsp = $this->set_response($message, REST_Controller::HTTP_NOT_FOUND);
}else{

    $hasil=[
        'result'=>"sukses"
    ];


    $rsp = $this->set_response( $hasil,REST_Controller::HTTP_CREATED);
}
```

Sebelum *request* dari klien di eksekusi, aplikasi terlebih dahulu melihat *role*-nya sebagai apa. Jika *role*-nya tidak sesuai, maka *request* tidak dapat di lanjutkan. Juga dapat ditambahkan nilai **randomID** yang disimpan pada setiap objek sesi.

API 2 Autentikasi Pengguna yang Rusak



Spesifik API	Dapat Dieksploitasi: 3	Prevalensi: 2	Dapat dideteksi: 2	Teknis: 3	Spesifik Bisnis
Autentikasi pada API merupakan mekanisme yang sangat kompleks dan membingungkan. Pengembang perangkat lunak dan praktisi keamanan dapat salah paham terkait batas autentikasi dan bagaimana cara mengimplementasikannya dengan baik. Selain itu, mekanisme autentikasi menjadi sasaran yang mudah bagi penyerang karena dapat dilihat oleh semua orang. Kedua poin tersebut menjadikan komponen autentikasi berpotensi rentan terhadap kebanyakan serangan.		Terdapat dua permasalahan utama, yaitu: 1) Kurangnya mekanisme perlindungan, *endpoint* API untuk autentikasi harus berbeda dari *endpoint* biasa serta menerapkan lapisan perlindungan yang lebih. 2) Mekanisme penerapan yang salah, yaitu menerapkan mekanisme autentikasi tanpa mempertimbangkan vektor serangan atau pada kasus yang tidak tepat (misal, mekanisme autentikasi pada IoT kurang tepat diterapkan pada aplikasi web).		Penyerang dapat mengontrol akun pengguna lain pada sistem, membaca data pribadi, dan melakukan tindakan sensitif atas nama mereka, seperti transaksi keuangan dan mengirim pesan pribadi.	

Detail Kerentanan

Endpoint dan alur autentikasi merupakan aset yang perlu dilindungi. "Lupa *password* / reset *password*" harus diperlakukan dengan cara yang sama seperti mekanisme autentikasi.

API dikatakan rentan jika:

- Mengizinkan *credential stuffing* dimana penyerang memiliki daftar *username* dan *password* yang valid.
- Mengizinkan penyerang melakukan serangan *brute force* pada *username* yang sama, tanpa mekanisme *captcha* / penguncian akun.
- Autentikasi lemah yang tidak mengikuti *best practice*.
- Mengizinkan *password* yang lemah.
- Menggunakan API *key* yang lemah yang tidak dirotasi.
- Mengirim detail autentikasi yang sensitif, seperti token autentikasi dan *password* di dalam URL.

- Tidak memvalidasi keaslian token.
- Menerima token JWT yang tidak ditandatangani (atau ditandatangani dengan lemah) ("alg":"none")/tidak memvalidasi tanggal kadaluwarsanya.
- Menggunakan *password* berupa teks biasa, terenkripsi, atau dengan *hash* yang lemah.
- Menggunakan kunci enkripsi yang lemah.

Contoh Skenario Serangan

Skenario 1

Credential stuffing (menggunakan daftar *username/password* yang diketahui), adalah serangan yang umum. Jika aplikasi tidak menerapkan perlindungan terhadap *credential stuffing*, maka aplikasi tersebut dapat digunakan sebagai pengujian *username* dan *password* (untuk mengetahui apakah kredensial tersebut valid).

Skenario 2

Penyerang dapat mencoba melakukan pemulihan *password* dengan melakukan *request* POST ke alamat (misal) **api/system/verification-codes** dengan mencantumkan *username* pada *request body*. Selanjutnya token SMS sebanyak 6 digit akan dikirimkan ke ponsel korban. Karena API tidak menerapkan kebijakan pembatasan laju, penyerang dapat menguji semua kemungkinan kombinasi menggunakan skrip terhadap **/api/system/verification-codes/{smsToken}** untuk menemukan token yang tepat hanya dalam beberapa menit.

Cara Mencegah

- Pastikan Anda mengetahui semua kemungkinan alur autentikasi ke API (seluler/web/tautan yang menerapkan autentikasi satu-klik, dan sebagainya).

- Koordinasikan dengan teknisi hal apa yang mungkin terlewatkan.
- Pahami mekanisme autentikasi yang Anda terapkan. Pastikan Anda memahami apa dan bagaimana penggunaannya. Sebagai catatan, OAuth bukanlah mekanisme autentikasi, begitu juga dengan API *key*.
- Jangan melakukan pengulangan pada autentikasi, pembuatan token, dan *password*.
- Pemulihan kredensial/lupa *password* harus diperlakukan seperti *endpoint* untuk *login,* harus disamakan dalam hal ketahanan terhadap *brute force*, pembatasan laju, dan *lockout protections*.
- Gunakan OWASP *Authentication Cheatsheet.*
- Jika memungkinkan gunakan autentikasi multi-faktor.
- Terapkan mekanisme anti *brute force* untuk mengurangi *credential stuffing*, *dictionary attack*, dan serangan *brute force* pada *endpoint* autentikasi. Mekanisme ini harus lebih ketat dibandingkan mekanisme pembatasan laju pada API.
- Terapkan mekanisme penguncian akun / *captcha* untuk mencegah *brute force* terhadap pengguna tertentu. Terapkan pemeriksaan terhadap *password* yang lemah.
- API *keys* tidak boleh digunakan untuk autentikasi pengguna, tapi hanya untuk autentikasi aplikasi/proyek klien.

Contoh Kasus Implementasi

```
$apikey="QX8PxBjIxD5RkfDnv0DyZVGXEOGfM72Ulj2nXYvwUcp33ydeL                    ";
$head=$this->input->request_headers();
$kunci= $head['Api-      '];

if($kunci==$apikey){

    $response = array(
        'result'=>"Sukses",
    );
    $rsp = $this->set_response( $response,REST_Controller::HTTP_OK);

}
else{
    $message=[
        'result'=>"Autentikasi Gagal"
    ];
    $rsp = $this->set_response($message, REST_Controller::HTTP_NOT_FOUND);
}
```

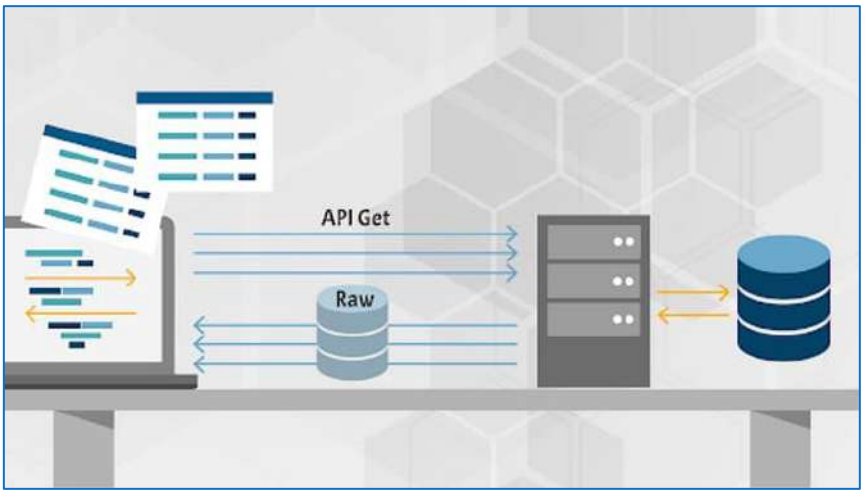
Mekanisme autentikasi menggunakan *Header* API *Key* dapat digunakan untuk mencegah kerentanan ini. Jika *request* dari klien tidak memiliki *Header* API yang sesuai, maka respon yang di dapatkan yaitu “Autentikasi Gagal”.

API 3 Paparan Data yang Berlebihan

Spesifik API	Dapat Dieksploitasi: 3	Prevalensi: 2	Dapat dideteksi: 2	Teknis: 2	Spesifik Bisnis
Eksploitasi dari paparan data yang berlebihan sebenarnya sederhana, dan biasanya dilakukan dengan cara *sniffing* pada *traffic* agar dapat menganalisis respons dari API, mencari data yang sensitif pada tampilan yang tidak seharusnya dikembalikan ke pengguna.		API mengandalkan klien untuk melakukan penyaringan data. Karena API digunakan sebagai sumber data, terkadang pengembang mencoba mengimplementasikannya dengan cara biasa tanpa memikirkan tentang sensitivitas data yang ada. *Tools* otomatis biasanya tidak dapat mendeteksi jenis kerentanan ini karena sulit untuk membedakan antara data yang sah yang dikembalikan dari API, dan data sensitif yang tidak boleh dikembalikan tanpa pemahaman yang mendalam tentang aplikasi.		Paparan Data yang Berlebihan umumnya menyebabkan bocornya data sensitif.	

Detail Kerentanan

API mengembalikan data yang sensitif pada klien berdasarkan desain yang ada. Data ini biasanya disaring pada sisi klien sebelum disajikan kepada pengguna. Penyerang dapat dengan mudah membaca *traffic* jaringan dan melihat data yang sensitif.



Contoh Skenario Serangan

Skenario 1

Misal tim *mobile* menggunakan **/api/articles/{articleId}/comments/{commentId}** pada tampilan artikel untuk merender metadata komentar. Dengan melihat *traffic* aplikasi *mobile*, penyerang mengetahui data sensitif lainnya yang berkaitan dengan komentar penulis. Implementasi pada API *endpoint* biasanya menggunakan *method* **toJSON()** pada model **User**, yang biasanya juga berisi data pribadi (misal NIK).

Skenario 2

Sistem pengawasan (CCTV) berbasis IoT memungkinkan administrator untuk membuat pengguna baru dengan *permission* yang berbeda. Seorang admin membuat akun pengguna untuk penjaga keamanan baru yang seharusnya hanya memiliki akses ke bangunan tertentu. Saat penjaga keamanan tersebut menjalankan aplikasinya, akan dilakukan API *call* ke: **/api/sites/111/cameras** untuk menerima data tentang kamera yang tersedia dan menampilkannya pada dasbor. Misal respon yang dikembalikan disusun dengan format : **{"id":"xxx","live_access_token":"xxxx bbbbb","building_id":"yyy"}**. Dalam kasus ini, GUI aplikasi klien hanya menunjukkan kamera yang dapat diakses oleh penjaga keamanan baru, namun respons API yang dikembalikan berisi daftar lengkap semua kamera yang ada di bangunan.

Cara Mencegah

- Jangan pernah mengandalkan sisi klien untuk menyaring data sensitif.
- Tinjau respons dari API untuk memastikan respons hanya berisi data yang sah.

- *Backend enggineers* harus memastikan siapa pengguna data sebelum mengekspos (mengaktifkan) *endpoint* API baru.
- Hindari menggunakan *method* umum seperti **to_json()** dan **to_string()**. Pilih dengan seksama respons apa yang akan dikembalikan dari suatu *request*.
- Klasifikasikan informasi sensitif yang disimpan pada aplikasi Anda dengan meninjau semua panggilan API yang mengembalikan informasi tersebut, lalu lihat apakah respons yang diberikan menimbulkan masalah keamanan.
- Terapkan mekanisme validasi respons sebagai lapisan keamanan tambahan.

Contoh Kasus Implementasi

```
    // $this->some_model->update_user( ... );
    $message = [
        'id' => 100, // Automatically generated by the model
        'name' => $this->post('name'),
        'email' => $this->post('email'),
        'message' => 'Added a resource'
    ];

    $this->set_response($message, REST_Controller::HTTP_CREATED); //
}
```

Untuk mencegah adanya kerentanan paparan data yang berlebihan, dapat dilakukan dengan mendefinisikan setiap respons dari API, dan tidak memberikan respons data secara utuh dari seluruh tabel pada basis data.

API 4 Kurangnya Sumber Daya & Pembatasan Laju

Spesifik API	Dapat Dieksploitasi: 2	Prevalensi: 3	Dapat dideteksi: 3	Teknis: 2	Spesifik Bisnis
Eksploitasi ini dilakukan menggunakan *request* API yang sederhana (tanpa autentikasi), yang dilakukan secara bersamaan, baik dilakukan dari satu komputer atau dengan menggunakan sumber daya komputasi awan (*cloud*).		Cukup umum untuk menemukan API yang tidak menerapkan pembatasan laju atau tidak diatur dengan benar.		Eksploitasi ini dapat menyebabkan DoS, membuat API tidak responsif atau bahkan mengganggu ketersediaan.	

Detail Kerentanan

Request API menggunakan sumber daya seperti jaringan, CPU, memori, dan penyimpanan. Jumlah sumber daya yang diperlukan untuk memenuhi *request* sangat tergantung pada input pengguna dan logika bisnis pada *endpoint*. Selain itu, *request* dari banyak klien API saling bersaing untuk mendapatkan sumber daya. API menjadi rentan jika setidaknya salah satu dari batas berikut tidak ada atau diatur secara tidak tepat (misal terlalu rendah/tinggi):

- Waktu tunggu eksekusi
- Memori maksimum yang dapat dialokasikan
- Jumlah deskriptor *file*
- Jumlah proses
- ukuran *payload request* (misal saat *upload*)
- Jumlah *request* per klien/*resource*
- Jumlah *record* per halaman yang dikembalikan dalam satu *request*

Contoh Skenario Serangan

Skenario 1

Penyerang mengunggah gambar yang besar dengan mengeluarkan *request* POST ke **/api/v1/images**. Ketika unggahannya selesai, API membuat beberapa *thumbnail* dengan ukuran berbeda-beda. Karena ukuran gambar yang besar, memori yang tersedia habis saat pembuatan *thumbnail* dan API menjadi tidak responsif.

Skenario 2

Misal terdapat aplikasi yang berisi daftar pengguna pada UI dengan batas 200 pengguna per halaman. Daftar pengguna diambil dari server menggunakan *query* berikut: **/api/users?page=1&size=100**. Selanjutnya seorang penyerang mengubah parameter **size** menjadi 200.000, sehingga menyebabkan masalah performa pada basis data. Sementara itu, API menjadi tidak responsif dan tidak dapat menangani *request* dari klien tersebut atau klien lainnya. Skenario yang sama juga dapat digunakan untuk memicu serangan *Integer Overflow* atau *Buffer Overflow*.

Cara Mencegah

- Tentukan pembatasan laju yang tepat.
- Gunakan *Docker* untuk memudahkan membatasi memori, CPU, jumlah *restart*, deskriptor *file*, dan proses.
- Terapkan batas seberapa sering klien dapat melakukan *request* API dalam jangka waktu yang ditentukan.
- Berikan notifikasi pada klien yang melebihi batas *request*.

- Tambahkan validasi pada sisi server untuk *string query* dan *request body* parameter, khususnya yang mengontrol jumlah *record* yang akan dikembalikan dalam respons.
- Tetapkan dan terapkan ukuran maksimum data pada semua parameter dan *payload* yang masuk, seperti panjang maksimum untuk *string* dan jumlah elemen maksimum dalam *array*.

Contoh Kasus Implementasi

```
function __construct()
{
    // Construct the parent class
    parent::__construct();

    $this->methods['data_post']['limit'] = 100; // 100 requests per hour per user/key
    $this->methods['users_delete']['limit'] = 50; // 50 requests per hour per user/key
}
```

Penggunaan limit dapat mencegah kerentanan ini. Limitasi dapat dilakukan pada setiap metode *request*, setiap pengguna, atau setiap alamat IP klien.

API 5 Otorisasi Tingkat Fungsi yang Rusak

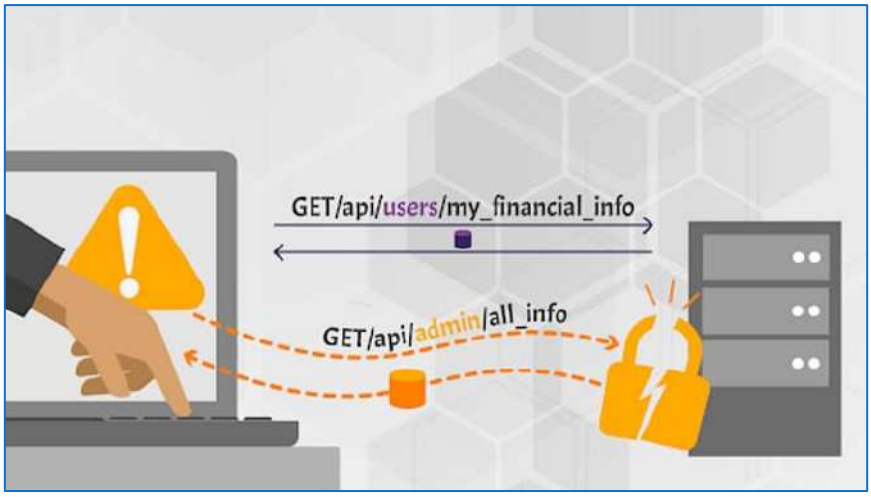
Spesifik API	Dapat Dieksploitasi: 3	Prevalensi: 2	Dapat dideteksi: 1	Teknis: 2	Spesifik Bisnis
Eksploitasi dilakukan penyerang dengan mengirim *request* API yang sah ke *endpoint* API, dimana seharusnya penyerang tidak memiliki akses tersebut. *Endpoint* API tersebut dapat terlihat oleh pengguna anonim atau reguler, maupun pengguna *non-privileged*. Kerentanan ini cukup mudah untuk ditemukan pada API karena API didesain lebih terstruktur, dan cara untuk mengakses fungsi tertentu lebih mudah diprediksi (misalnya, mengganti **HTTP** *method* dari **GET** ke **PUT**, atau mengubah string "**users**" di **URL** menjadi "**admin**").		Pemeriksaan otorisasi untuk suatu fungsi atau sumber daya biasanya dikelola melalui konfigurasi, atau juga pada level kode. Penerapan pemeriksaan otorisasi yang tepat cukup sulit dilakukan karena aplikasi modern berisi banyak jenis *role* atau kelompok, dan hierarki pengguna yang kompleks (misalnya, sub-pengguna, atau pengguna dengan lebih dari satu *role*).		Kerentanan ini memungkinkan penyerang untuk memperoleh akses yang tidak sah. Akun/fungsi admin adalah target utama dari serangan ini.	

Detail Kerentanan

Cara untuk menemukan masalah otorisasi tingkat fungsi yang rusak adalah dengan melakukan analisis mendalam terhadap mekanisme otorisasi, dengan mempertimbangkan hierarki pengguna, peran atau grup yang berbeda dalam aplikasi, serta menanyakan hal-hal berikut:

- Bisakah pengguna biasa mengakses *endpoint* admin?
- Dapatkah pengguna melakukan tindakan sensitif (seperti melakukan pembuatan, modifikasi, atau penghapusan) yang seharusnya tidak bisa dilakukan, hanya dengan mengubah **HTTP** *method* (misalnya dari **GET** ke **DELETE**)?
- Bisakah pengguna dari grup X mengakses fungsi yang seharusnya hanya untuk pengguna dari grup Y, hanya dengan menebak parameter dan *endpoint* **URL** (misal **/api/v1/users/export_all)**?

Jangan berasumsi bahwa *endpoint* API bersifat reguler atau admin hanya berdasarkan URL. Developer dapat memilih untuk mengekspos *endpoint* admin pada *path* khusus, misal **api/admins**, atau sangat umum untuk menemukan *endpoint* admin pada *path* yang sama dengan *endpoint* reguler, misal **api/users**.



Contoh Skenario Serangan

Skenario 1

Misal suatu aplikasi mengizinkan registrasi hanya kepada pengguna yang diundang, saat proses tersebut, aplikasi melakukan *request* API **GET /api/invites/{invite_guid}.** Respon yang diberikan berupa **JSON** berisi detail undangan, termasuk peran pengguna dan *email* pengguna.

Selanjutnya seorang penyerang mencoba menduplikasi *request* tersebut dan memanipulasi **HTTP** *method* dan *endpoint* menjadi **POST /api/invites/new.** *Endpoint* API tersebut seharusnya hanya dapat diakses oleh admin menggunakan konsol admin, dan dalam kasus ini tidak menerapkan pemeriksaan otorisasi tingkat fungsi. Akhirnya, penyerang mengeksploitasi kerentanan tersebut dan mengirim sendiri undangan untuk membuat akun admin dengan *request*:

POST /api/invites/new

{"email":"hugo@malicious.com","role":"admin"}

Skenario 2

Misal suatu API berisi *endpoint* yang hanya boleh diakses oleh administrator: **GET /api/admin/v1/user/all.** *Endpoint* ini memberikan respon yaitu detail semua pengguna

aplikasi dan tidak menerapkan pemeriksaan fungsi tingkat otorisasi. Seorang penyerang yang mempelajari struktur API tersebut dapat menebak dan mengakses *endpoint* ini, sehingga berhasil mendapatkan seluruh data pengguna dari aplikasi.

Cara Mencegah

Aplikasi harus memiliki modul otorisasi yang konsisten dan mudah dianalisis yang diperoleh dari semua fungsi bisnis. Sering kali, perlindungan tersebut didapat dari komponen di luar kode aplikasi.

- Aplikasi didesain untuk menolak semua akses secara *default*, dan membutuhkan peran yang eksplisit untuk dapat mengakses ke setiap fungsi.
- Tinjau *endpoint* API untuk mendeteksi kerentanan fungsi tingkat otorisasi, dengan mempertimbangkan logika bisnis aplikasi dan hierarki grup.
- Pastikan bahwa semua kontrol admin didapat dari *administrative abstract controller* yang mengimplementasikan pemeriksaan otorisasi berdasarkan grup/peran pengguna.
- Pastikan bahwa fungsi administratif di dalam *regular controller* mengimplementasikan pemeriksaan otorisasi berdasarkan grup dan peran pengguna.

Contoh Kasus Implementasi

```
if($this->session->userdata('role')!="admin"){
    $message=[
        'result'=>"gagal"
    ];

    $rsp = $this->set_response($message, REST_Controller::HTTP_NOT_FOUND);
}else{

    $hasil=[
        'result'=>"sukses"
    ];


    $rsp = $this->set_response( $hasil,REST_Controller::HTTP_CREATED);
}
```

Sebelum *request* di eksekusi, maka aplikasi melihat *role*-nya terlebih dahulu. Jika *role*-nya tidak sesuai maka *request* tidak dapat di lanjutkan. Respons akan diberikan sesuai *role* dari klien.

API 6 *Mass Assignment*

Spesifik API	Dapat Dieksploitasi: 2	Prevalensi: 2	Dapat dideteksi: 2	Teknis: 2	Spesifik Bisnis
Eksploitasi ini membutuhkan pemahaman tentang logika bisnis aplikasi, relasi objek, dan struktur API. Jenis eksploitasi ini lebih mudah diterapkan pada API karena secara desain API menampilkan dasar dari implementasi aplikasi bersama dengan nama propertinya.		Kerangka kerja modern mendorong pengembang untuk menggunakan fungsi yang secara otomatis menghubungkan (mengikat) input dari klien ke dalam variabel kode dan objek internal. Penyerang dapat menggunakan metodologi ini untuk memperbarui atau menimpa properti objek tertentu yang tidak sengaja diekspos oleh pengembang.		Eksploitasi ini dapat menyebabkan *privilege escalation*, gangguan data, penerobosan terhadap mekanisme keamanan, dan lain sebagainya.	

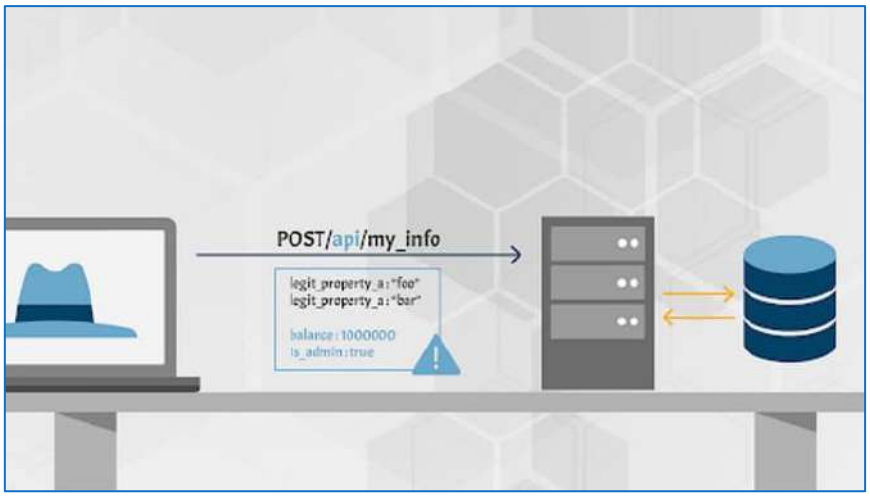
Detail Kerentanan

Objek dalam aplikasi modern dapat berisi banyak properti. Beberapa properti dapat diperbarui langsung oleh klien (seperti **user.first_name** or **user.adrdress**) dan beberapa properti seharusnya tidak dapat diubah klien (seperti **user.is_vip flag**).

Endpoint API dikatakan rentan jika secara otomatis dapat mengubah parameter klien menjadi properti objek internal, tanpa mempertimbangkan sensitivitas dan tingkat eksposur dari properti. Hal ini memungkinkan penyerang untuk memperbarui properti dari objek yang seharusnya tidak dapat diakses.

Contoh properti yang sensitif:

- Properti terkait perizinan: **user.is_admin, user.is_vip** hanya boleh diatur oleh admin.
- Properti yang bergantung pada proses: **user.cash** hanya boleh diatur secara internal setelah dilakukan verifikasi pembayaran.
- Properti internal: **article.created_time** hanya boleh diatur secara internal oleh aplikasi.



Contoh Skenario Serangan

Skenario 1

Misal suatu aplikasi memberi pengguna pilihan untuk mengedit informasi untuk profil mereka. Saat proses edit profil ini, aplikasi melakukan panggilan API: **PUT /api/v1/users/me** dengan objek **JSON** yang sah sebagai berikut:

{"user_name":"inons","age":24}

Saat dilakukan *request* **GET /api/v1/users/me**, maka akan direspon dengan menyertakan properti tambahan *(credit_balance)*:

{"user_name":"inons","age":24,"credit_balance":10}.

Memanfaatkan kerentanan ini, penyerang mengulang panggilan API **PUT /api/v1/users/me** dengan muatan objek JSON berikut:

{"user_name":"attacker","age":60,"credit_balance":99999}

Karena *endpoint* rentan terhadap *mass assignment*, penyerang berhasil menerima kredit tanpa harus membayar.

Skenario 2

Suatu portal berbagi video memungkinkan pengguna mengunggah konten dan mengunduh konten dalam berbagai format. Seorang penyerang yang mengeksplorasi API menemukan bahwa *endpoint* **GET /api/v1/videos/{video_id}/meta_data** mengembalikan objek **JSON** berisi properti video. Salah satu propertinya adalah **"mp4_conversion_params":"-v codec h264"**, yang menunjukkan bahwa aplikasi menggunakan perintah *shell* untuk mengonversi video.

Penyerang juga menemukan bahwa *endpoint* **POST /api/v1/videos/new** rentan terhadap *mass assignment* dan memungkinkan klien untuk mengatur properti apapun dari objek video. Lalu penyerang menyisipkan kode berbahaya berikut di dalam salah satu properti video: **"mp4_conversion_params":"-v codec h264 && format C:/"**. Kode *shell* **format C:/** akan ikut dijalankan saat penyerang mengunduh video dengan format MP4**.**

Cara Mencegah

- Jika memungkinkan, hindari menggunakan fungsi yang secara otomatis mengikat input klien ke dalam variabel kode atau objek internal.
- Buat *white list* properti yang dapat diperbarui oleh klien.
- Buat *black list* properti yang tidak boleh diakses oleh klien.
- Jika memungkinkan, tentukan secara eksplisit dan terapkan skema untuk muatan data input.

Contoh Kasus Implementasi

```
$data=array(
    'nama'=>$this->post('nama'),
    'jns_identitas'=>$this->post('jns_identitas'),
    'nik'=>$this->post('nik'),
    'tgl_lahir'=>$this->post('tgl_lahir'),
    'usia_thn'=>$this->post('usia_thn'),
    'usia_bln'=>$this->post('usia_bln'),
    'jns_kelamin'=>$this->post('jns_kelamin'),
    'no_hp'=>$this->post('no_hp'),
    'alamat_ktp'=>$this->post('alamat_ktp'),
```

Definisikan setiap input yang masuk melalui API sehingga dapat mencegah kerentanan *Mass Assignment*.

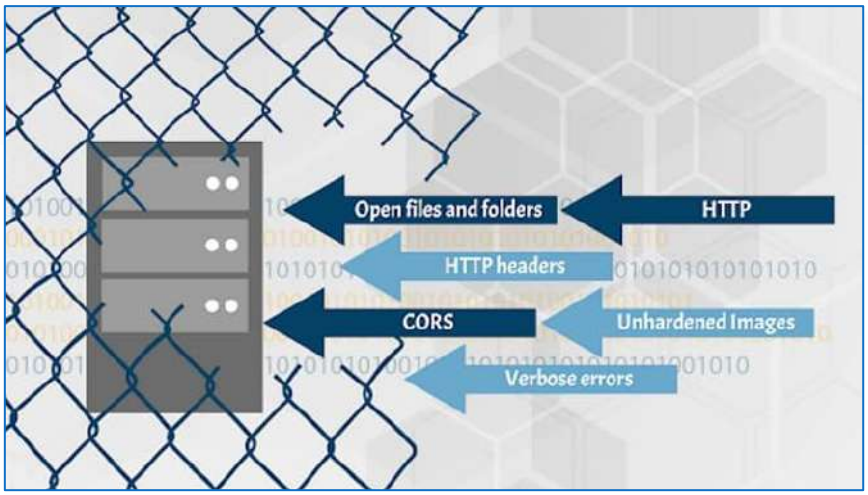
API 7 Kesalahan Konfigurasi Keamanan

Spesifik API	Dapat Dieksploitasi: 3	Prevalensi: 3	Dapat dideteksi: 3	Teknis: 2	Spesifik Bisnis
Penyerang mencoba mencari kerentanan yang belum di-*patch*, *endpoint* umum, atau *file* dan direktori yang belum diamankan, untuk mendapatkan akses atau pengetahuan terkait sistem.		Kesalahan konfigurasi keamanan dapat terjadi pada semua tingkatan API, dari level jaringan hingga level aplikasi. *Automated tools* tersedia untuk mendeteksi dan mengeksploitasi kesalahan konfigurasi seperti *service* yang tidak diperlukan dan *legacy option.*		Kesalahan konfigurasi keamanan dapat menyebabkan bocornya data sensitif hingga *server compromise*.	

Detail Kerentanan

API menjadi rentan apabila:

- *Security hardening* tidak diterapkan di lapisan aplikasi, atau izin tidak dikonfigurasi dengan benar pada layanan *cloud*.
- *Patch* keamanan tidak diperbarui, atau sistem sudah *out of date*.
- Terdapat fitur yang seharusnya tidak perlu diaktifkan (seperti HTTP).
- Tidak menggunakan *Transport Layer Security* (TLS).
- Petunjuk keamanan tidak diberikan kepada klien (misal *Security Header*).
- Tidak ada Kebijakan *Cross-Origin Resource Sharing* (CORS) atau tidak diatur dengan benar.
- Pesan *error* termasuk *stack traces* atau informasi sensitif lainnya terekspos.



Contoh Skenario Serangan

Skenario 1

Penyerang menemukan *file .bash_history* pada direktori *root server*, yang berisi *command* yang digunakan oleh tim DevOps untuk mengakses API:

$ curl -X GET 'https://api.server/endpoint/' -H 'authorization: Basic Zm9vOmJhcg=='

Penyerang juga dapat menemukan *endpoint* baru pada API yang hanya digunakan oleh tim DevOps dan tidak didokumentasikan.

Skenario 2

Untuk menargetkan *service* tertentu, penyerang menggunakan mesin pencari untuk mencari server/komputer yang dapat diakses langsung dari internet. Misal penyerang dapat menemukan *host* yang menjalankan sistem manajemen basis data populer, dan menggunakan *port default*. *Host* tersebut juga menggunakan konfigurasi *default*, dengan autentikasi yang secara *default* dinonaktifkan, sehingga penyerang berhasil memperoleh akses ke jutaan *record*, data pribadi, dan data autentikasi.

Cara Mencegah

Siklus hidup API harus mencakup:

- Proses *hardening* berulang yang mengarah pada penerapan yang cepat dan mudah untuk lingkungan sistem yang aman (terkunci).
- Peninjauan dan pembaharuan konfigurasi di seluruh lapisan API. Tinjauan tersebut harus mencakup: *file* orkestrasi, komponen API, dan layanan *cloud* (misal *bucket permissions* S3).

- Saluran komunikasi yang aman untuk semua akses interaksi API ke aset statis (misalnya: Gambar).
- Proses otomasi untuk terus menilai efektivitas konfigurasi dan pengaturan di semua lingkungan.

Selanjutnya:

- Untuk mencegah *exception traces* dan informasi berharga/sensitif dikirim kembali ke penyerang, tentukan dan terapkan semua skema *response payload* API termasuk respons *error*.
- Pastikan API hanya dapat diakses oleh HTTP *verbs* yang ditentukan. Semua HTTP *verbs* lainnya harus dinonaktifkan (misal HEAD).
- API yang diakses dari browser klien (misal WebApp *front-end*) harus menerapkan kebijakan *Cross-Origin Resource Sharing* (CORS) yang tepat.

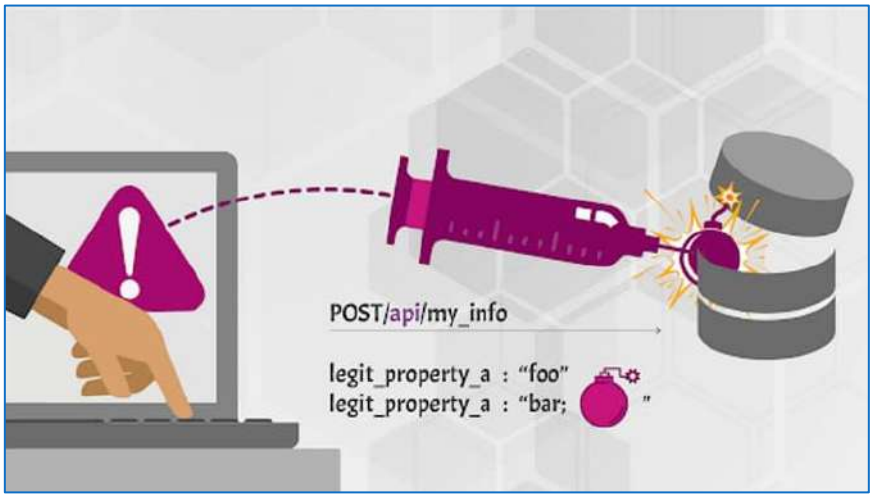
API 8 Injeksi

Penyerang	Vektor Serangan	Kelemahan Keamanan		Dampak	
Spesifik API	**Dapat Dieksploitasi: 3**	**Prevalensi: 2**	**Dapat dideteksi: 3**	**Teknis: 3**	**Spesifik Bisnis**
Penyerang dapat membuat API *calls* yang mengandung SQL, NoSQL, LDAP, OS, atau *command* lainnya yang dapat dieksekusi oleh API atau *backend*.		Kerentanan injeksi sangat umum dan sering ditemukan dalam *query* SQL, LDAP, atau NoSQL, OS *command*, parser XML, dan ORM. Kerentanan ini dapat ditemukan saat dilakukan peninjauan terhadap kode sumber aplikasi. Penyerang dapat menggunakan *tools vulnerability scanner* untuk menemukan kerentanan tersebut.		Kerentanan injeksi dapat menyebabkan bocor atau hilangnya data. Selain itu juga dapat menyebabkan DoS, atau pengambilalihan *host* secara penuh.	

Detail Kerentanan

API menjadi rentan apabila:

- Data dari klien tidak divalidasi, difilter, atau dibersihkan (di-sanitasi) oleh API.
- Data dari klien langsung digunakan ke *query* SQL/NoSQL/LDAP, OS *command*, parser XML, dan *Object Relational Mapping* (ORM)/*Object Document Mapper* (ODM).
- Data yang berasal dari sistem eksternal (misal sistem terintegrasi) tidak divalidasi, difilter, atau di-sanitasi oleh API.



Contoh Skenario Serangan

Skenario 1

Misal suatu *firmware* dari *parental control device* menyediakan *endpoint* **/api/CONFIG/restore** yang digunakan untuk mengirim **appId** sebagai suatu parameter. Dengan menggunakan *decompiler* dan metode tertentu, penyerang dapat mengetahui bahwa nilai **appId** langsung dieksekusi oleh sistem tanpa dilakukan sanitasi:

```
snprintf(cmd, 128, "%srestore_backup.sh /tmp/postfile.bin %s %d",
      "/mnt/shares/usr/bin/scripts/", appid, 66);
sistem (cmd);
```

Berikut adalah *command* yang memungkinkan penyerang untuk mematikan perangkat dengan kerentanan *firmware* yang sama:

```
$ curl -k "https://${deviceIP}:4567/api/CONFIG/restore" -F
'appid=$(/etc/pod/power_down.sh)'
```

Skenario 2

Misal terdapat aplikasi *booking* dengan fungsi dasar CRUD. Penyerang diasumsikan berhasil mengidentifikasi bahwa injeksi NoSQL dapat dilakukan melalui parameter **bookingId** dalam *request* hapus *booking*. Berikut contoh *request*-nya:

```
DELETE/api/bookings?bookingId=678
```

Server API menggunakan fungsi berikut untuk menangani *request* penghapusan *booking*:

```
router.delete('/bookings', async function (req, res, next) {
  try {
    const deletedBooking = await Bookings.findOneAndRemove({_id' :
    req.query.bookingId});
    res.status(200);
 }  catch (err) {
    res.status(400).json({
      error: 'Unexpected error occured while processing a request'
    });
   }
});
```

Selanjutnya penyerang mengintersepsi *request* dan mengubah parameter **bookingId** seperti yang ditunjukkan di bawah. Dalam kasus ini, penyerang dapat menghapus **bookingId** dari pengguna lain:

```
DELETE/api/bookings?bookingId[$ne]=678
```

Cara Mencegah

Secara umum, cara untuk mencegah injeksi adalah dengan memisahkan antara penyimpanan data dengan *command* dan *query*.

- Melakukan validasi data menggunakan suatu pustaka yang terpercaya dan dikelola dengan baik.
- Memvalidasi, memfilter, dan membersihkan data yang diterima dari klien, atau data lain yang berasal dari sistem terintegrasi lainnya.
- Karakter khusus harus diloloskan menggunakan sintaks khusus untuk translasi target.
- Menggunakan API yang aman dan menyediakan antarmuka dengan parameter.
- Selalu membatasi jumlah *records* yang dikembalikan ke klien untuk mencegah kebocoran banyak data jika terjadi injeksi.
- Validasi data yang masuk menggunakan filter yang memadai dengan hanya mengizinkan nilai valid untuk setiap parameter input.
- Tentukan tipe data dan pola yang ketat untuk semua parameter *string*.

Contoh Kasus Implementasi

Kerentanan injeksi dapat di cegah dengan penerapan filter dan sanitasi pada setiap input. Sebagai contoh penerapannya yaitu dengan menggunakan *escape* pada saat *query* ke dalam basis data.

```
public function simpan_data($api)
{
    $sql = "INSERT INTO table (title) VALUES(".$this->db->escape($api).")";
```

Selain *escape*, metode yang dapat dilakukan yaitu dengan penerapan *Query binding*.

```
$sql = "SELECT * FROM some_table WHERE id = ? AND status = ? AND author = ?";
$this->db->query($sql, array(3, 'live', 'Rick'));
```

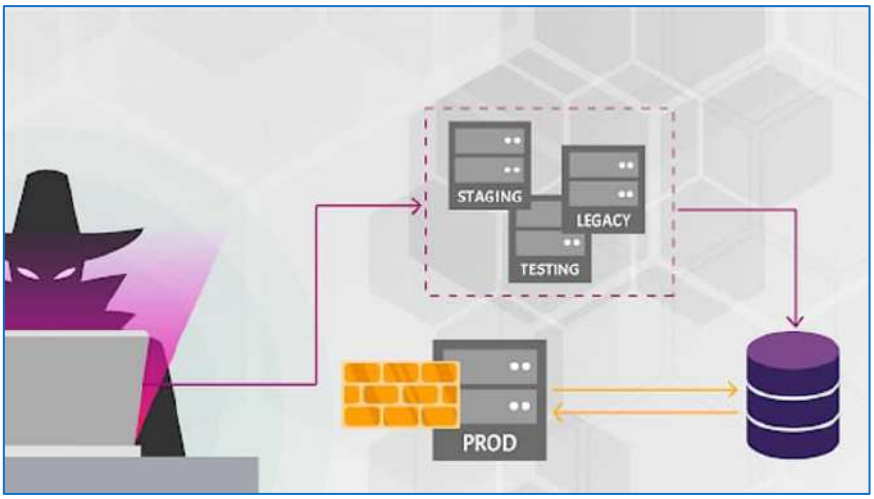
API 9 Manajemen Aset yang Tidak Benar

Spesifik API	Dapat Dieksploitasi: 3	Prevalensi: 3	Dapat dideteksi: 2	Teknis: 2	Spesifik Bisnis
Kondisi dimana penyerang menemukan versi *non-production* dari API (seperti versi contoh, *staging*, testing, beta, atau versi lama) yang tidak diamankan seperti versi *production.* Lalu penyerang memanfaatkan API versi *non-production* tersebut untuk melakukan serangan.		Dokumentasi API yang tidak diperbarui membuat kerentanan menjadi sulit untuk ditemukan dan diperbaiki. Kurangnya inventarisasi aset dan strategi *retire* berdampak dijalankannya sistem yang tidak di-*patch* dan mengakibatkan kebocoran data sensitif.		Penyerang memperoleh akses ke data sensitif, atau bahkan mengambil alih server melalui versi API lama yang belum di-*patch* yang terhubung ke basis data yang sama.	

Detail Kerentanan

API menjadi rentan apabila:

- Tujuan dibuatnya *host* API tidak jelas. Pengguna juga sulit mengidentifikasi jawaban dari pertanyaan-pertanyaan berikut:
  - Lingkungan apa yang sedang menjalankan API (misal *production*, *staging*, *testing*, pengembangan)?
  - Siapa yang memiliki akses jaringan ke API (misal publik, internal, mitra)?
  - API versi berapa yang sedang berjalan?
  - Data apa yang dikumpulkan dan diproses oleh API (misal NIK)?
  - Bagaimana aliran datanya?
- Tidak ada dokumentasi API, atau dokumentasi yang ada tidak diperbarui.
- Tidak ada *retirement plan* untuk setiap versi API.
- Tidak ada inventarisasi *host*.
- Tidak ada inventarisasi *service* yang terintegrasi, baik pihak pertama atau ketiga.
- Versi API lama yang tetap berjalan tanpa di-*patch*.



Contoh Skenario Serangan

Skenario 1

Misal terdapat aplikasi yang membiarkan API versi lama tetap berjalan **(api.someservice.com/v1)**, tidak diproteksi, dan tetap memiliki akses ke basis data pengguna.

Lalu seorang penyerang sedang melakukan serangan pada salah satu aplikasi terbaru yang dirilis, dan menemukan alamat API **(api.someservice.com/v2)**. Dengan mencoba mengganti v2 menjadi v1 di URL, penyerang dapat mengakses API lama yang tidak diproteksi, dan berhasil mendapatkan informasi sensitif pengguna.

Skenario 2

Suatu aplikasi jejaring sosial menerapkan mekanisme pembatasan laju untuk memblokir penyerang yang melakukan *brute-force* token reset *password*. Mekanisme ini tidak diterapkan dalam kode API, melainkan dalam komponen terpisah antara klien dan API **(www.socialnetwork.com).** Seorang penjahat menemukan *host* API beta **(www.mbasic.beta.socialnetwork.com)** yang menjalankan API yang sama, termasuk fitur reset *password*, namun belum menerapkan mekanisme pembatasan laju. Dengan begitu, penjahat dapat melakukan reset *password* pengguna mana pun menggunakan *brute force* sederhana untuk menebak 6 digit token.

Cara Mencegah

- Inventarisi semua *host* API dan mendokumentasikan aspek penting dari masing-masing *host*, dengan fokus pada lingkungan API (misal *production*, *staging*, *testing*, pengembangan), siapa yang memiliki akses jaringan ke *host* (misalnya, publik, internal, mitra) dan suatu versi API.
- Inventarisi *services* yang ada dan dokumentasikan aspek-aspek penting seperti perannya dalam sistem, data yang dipertukarkan (data *flow*), dan sensitivitas data.
- Dokumentasikan semua aspek API seperti autentikasi, *error*, pengalihan, pembatasan laju, kebijakan dan *endpoint cross-origin resource sharing* (CORS), termasuk parameter, *request*, dan respons.
- Buat dokumentasi API secara otomatis menggunakan *standard* tertentu.
- Sediakan dokumentasi API hanya bagi yang berwenang untuk menggunakan API.
- Gunakan perimeter keamanan eksternal seperti *firewall* untuk semua versi API yang diekspos, bukan hanya untuk versi produksi terbaru.
- Hindari menggunakan data *production* pada penerapan API *non-production*. Atau jika terpaksa, *endpoint* tersebut harus mendapat perlakuan keamanan yang sama dengan versi *production*.
- Saat versi terbaru API melakukan peningkatan keamanan, lakukan analisis risiko untuk tindakan mitigasi terhadap versi API yang lama: misal, apakah mungkin untuk melakukan peningkatan keamanan tanpa merusak kompatibilitas API, atau apakah perlu menghapus API versi lama dan memaksa semua klien untuk pindah ke versi baru.

API 10 *Logging* dan Pemantauan Tidak Memadai

Penyerang → Vektor Serangan		Kelemahan Keamanan		Dampak	
Spesifik API	**Dapat Dieksploitasi: 2**	**Prevalensi: 3**	**Dapat dideteksi: 1**	**Teknis: 2**	**Spesifik Bisnis**
Penyerang memanfaatkan kurangnya *logging* dan pemantauan untuk menyerang sistem tanpa terdeteksi.		Tanpa adanya *logging* dan pemantauan, hampir tidak mungkin untuk melacak aktivitas yang mencurigakan dan melakukan respon secara tepat waktu.		Sistem sepenuhnya bocor oleh penyerang karena tidak ada visibilitas akan aktivitas jahat yang sedang berlangsung.	

Detail Kerentanan

API menjadi rentan apabila:

- Sistem tidak menghasilkan log apa pun, level *logging* tidak diatur dengan benar, atau log tidak detail.
- Integritas log tidak terjamin (misal terjadi injeksi Log).
- Log tidak dipantau secara kontinu.
- Infrastruktur API tidak dipantau secara kontinu.

Contoh Skenario Serangan

Skenario 1

Misal kunci akses API admin suatu perusahaan bocor di publik. Perusahaan tersebut diberitahu melalui *email* tentang potensi kebocoran, namun membutuhkan waktu lebih dari 48 jam untuk merespon insiden tersebut. Selama waktu tersebut, kunci akses API mungkin telah digunakan untuk mengakses data sensitif. Karena *logging* yang tidak memadai, perusahaan tersebut tidak dapat mengetahui data apa saja yang diakses oleh penjahat.

Skenario 2

Misal terdapat platform berbagi video yang terkena serangan *credential stuffing*. Meskipun *failed logins* telah dicatat dalam log, namun tidak ada peringatan *(alert)* yang muncul selama serangan berlangsung. Perusahaan baru mengetahui serangan tersebut melalui keluhan dari pengguna, sehingga akhirnya dilakukan analisis log API untuk melihat serangan. Dengan kejadian ini, perusahaan harus membuat pengumuman ke publik untuk meminta pengguna melakukan reset *password*, dan melaporkan serangan tersebut kepada pihak berwenang.

Cara Mencegah

Siklus hidup API harus mencakup:

- Catat semua upaya autentikasi yang gagal, akses yang ditolak, dan kesalahan validasi input ke dalam log.
- Log harus ditulis menggunakan format yang kompatibel dengan *log management solution*, dan harus mencakup detail untuk mengidentifikasi penyerang.
- Log harus diperlakukan sebagai data sensitif, dan integritasnya harus dijamin saat disimpan dan dikirimkan.
- Konfigurasikan sistem pemantauan untuk terus memantau infrastruktur, jaringan, dan fungsional API.
- Gunakan *Security Information and Event Management* (SIEM) untuk mengumpulkan dan mengelola log dari semua komponen API *stack* dan API *host*.
- Konfigurasi dasbor dan *alert* untuk mendeteksi aktivitas mencurigakan sehingga dapat direspon lebih cepat.

CATATAN UNTUK PENGEMBANG

Tantangan seorang pengembang adalah membuat, memperbaiki dan memelihara perangkat lunak yang aman, termasuk aplikasi berbasis *microservice* atau API. Edukasi dan kesadaran keamanan adalah faktor kunci dalam membuat perangkat lunak yang aman. Selain dua hal tersebut, sisanya bergantung pada pembangunan dan penggunaan proses keamanan berlapis dan kontrol keamanan standar.

OWASP telah menyediakan banyak referensi gratis dalam menangani keamanan sejak awal proyek aplikasi. Berikut adalah beberapa referensi dari OWASP yang dapat menjadi acuan dan telah dikelompokkan menjadi beberapa bagian.

Edukasi	Pengembang dapat memulai dengan membaca **OWASP *Education Project materials*,** sesuai dengan profesi dan minat. Untuk pembelajaran *hands-on*, terdapat **crAPI - *Completely Radiculous* API**. Lalu untuk latihan WebAppSec dapat menggunakan **OWASP DevSlop Pixi Module.** Terdapat juga sesi pelatihan melalui **OWASP AppSec *Conference***.
Persyaratan Keamanan (*Security Requirements*)	OWASP menyarankan **OWASP *Application Security Verification Standard* (ASVS)** sebagai panduan untuk mengatur persyaratan keamanan. Jika dilakukan secara *outsourcing*, dapat menggunakan **OWASP *Secure Software Contract Annex***, yang disesuaikan dengan hukum dan peraturan setempat.
Arsitektur Keamanan	**OWASP *Prevention Cheat Sheets*** dapat menjadi panduan awal dalam merancang keamanan arsitektur. Selain itu terdapat juga **REST *Security Cheat Sheet*** dan **REST *Assessment Cheat Sheet.***
Kontrol Keamanan Standar	**OWASP *Proactive Controls*** dapat memberi gambaran tentang kontrol keamanan apa yang dapat digunakan dalam proyek aplikasi. OWASP juga menyediakan beberapa pustaka dan *tools* yang dapat digunakan, misal untuk kontrol validasi.
Siklus Hidup Pengembangan Perangkat Lunak yang Aman	**OWASP *Software Assurance Maturity Model* (SAMM)** dapat digunakan untuk membantu proses membuat API. Beberapa referensi OWASP lainnya juga dapat membantu saat fase pengembangan API, seperti **OWASP *Code Review Project***.

CATATAN UNTUK DEV-SEC-OPS

Karena pentingnya API dalam arsitektur aplikasi modern, membangun API yang aman menjadi sangat penting. Keamanan harus menjadi bagian dari keseluruhan siklus hidup pengembangan aplikasi. *Scanning* dan *penetration testing* yang dilaksanakan setiap tahun dianggap tidak cukup untuk menjamin keamanan.

Tim DevSecOps harus bergabung sejak awal pengembangan aplikasi, memfasilitasi pengujian keamanan berkelanjutan di seluruh siklus hidup pengembangan perangkat lunak. Tujuan tim ini adalah untuk memaksimalkan alur pengembangan melalui otomatisasi keamanan, tanpa mempengaruhi kecepatan pengembangan.

DevSecOps manifesto dapat ditinjau melalui https://www.devsecops.org/.

Pahami Model Ancaman	**OWASP *Application Security Verification* Standard (ASVS)** dapat menjadi panduan untuk *threat model*, dan **OWASP *Testing Guide*** sebagai masukan. Melibatkan tim pengembang dapat membantu mereka lebih sadar akan keamanan.
Memahami *Software Development Life Cycle* (SDLC)	Bergabunglah dengan tim pengembang agar lebih memahami Siklus Hidup Pengembangan Perangkat Lunak. Kontribusi DevSecOps pada pengujian keamanan harus kompatibel dengan *people*, *processes*, dan *tools*. Semua orang harus sepakat dengan prosesnya, sehingga tidak terjadi gesekan yang tidak perlu.
Strategi Pengujian	Karena pekerjaan DevSecOps seharusnya tidak memengaruhi kecepatan pengembangan, Anda harus bijak dalam memilih pendekatan terbaik (sederhana, tercepat, paling akurat) untuk memverifikasi persyaratan keamanan. **OWASP *Security Knowledge Framework*** dan **OWASP *Application Security Verification Standard*** dapat menjadi acuan untuk persyaratan keamanan fungsional dan non fungsional. Referensi dan *tools* lainnya dapat dilihat di https://www.devsecops.org/.
Mencapai Cakupan dan Akurasi	DevSecOps adalah jembatan antara pengembang dan tim operasi. Anda tidak hanya harus fokus pada fungsionalitas, tetapi juga orkestrasi. Bekerjalah dekat dengan tim

	pengembang dan operasi sejak awal sehingga Anda dapat mengoptimalkan waktu dan upaya. Anda harus fokus agar keamanan esensial diverifikasi terus menerus.
Komunikasikan Temuan dengan Jelas	Laporkan segala temuan dengan tepat waktu, dalam format yang digunakan (dimengerti) tim pengembang. Bergabunglah dengan tim pengembang untuk mengatasi temuan tersebut. Lakukan *sharing knowledge*, dan gambarkan dengan jelas kelemahan serta dampaknya, termasuk skenario serangan.

BADAN SIBER
DAN SANDI
NEGARA

PUSAT PENGKAJIAN DAN PENGEMBANGAN
TEKNOLOGI KEAMANAN SIBER DAN SANDI

Agustus 2021 | Badan Siber dan Sandi Negara

Tel: +62217805814 Email: humas@bssn.go.id https://www.bssn.go.id