NIST Special Publication 800-204

마이크로서비스 기반 애플리케이션 시스템 보안 전략

Security Strategies for Microservices-based Application Systems

권 리

이 문서는 NIST(National Institue of Standards and Technonolgy)에서 발행한 "NIST SP(Special Publication) 800-204, Security Strategies for Microservices-based Application Systems"을 한국어로 번역한 결과물이다. 이 문서와 관련된 모든 권리는 NIST에 있다.

- 저자 : Ramaswamy Chandramouli, NIST
- 번역 : Youngeun Moon, Blackfalcon Security

개 요

마이크로서비스 아키텍처를 사용하여 애플리케이션 시스템을 개발하는 사례가 증가하고 있다. 마이크로서비스 아키텍처는 코드의 양의 적어, 개발-테스트-배포가 빠르다. 또한, 플랫폼이 개발팀에 독립적이고, 각 컴포넌트에 대한 개별적인 확장도 가능하여 플랫폼의 최적화도 용이하다. 마이크로서비스 사이의 통신을 위해 일반적으로 API를 사용하는데, 다수의 컴포넌트 사이의 복잡한 상호작용을 지원하기 위해 몇몇 핵심 기능이 필요하다. 핵심 기능이란, 인증과 접근 관리, 서비스 검색, 보안 통신 프로토콜, 보안 모니터링, 가용성/복원력 개선 기술(예 : 서킷브레이커[1]), 부하 분산, 스로틀링(throttling)[2], 신규 서비스 도입 시 무결성 보장 기술, 세션 처리 등을 말한다. 이러한 핵심 기능은 API 게이트웨이 및 서비스 메시와 같은 아키텍처 프레임워크에 포함되어 있을 수 있다. 이 문서의 목적은 각 핵심 기능을 구현하는데 사용할 수 있는 다양한 방법과 아키텍처 프레임워크의 다양한 설정 방법에 대해 분석하고, 마이크로서비스에 대한 위협에 대응할 수 있는 보안 전략을 개발하며, 마이크로서비스 기반 애플리케이션의 전반적인 보안 프로파일을 개선하는데 있다.

---

[1] 서킷브레이커란, 마이크로서비스 기반 애플리케이션의 가용성을 높이기 위한 기능으로 4.5.1절에서 자세히 설명한다.

[2] 스로트링이란, 마이크로서비스 기반 애플리케이션의 가용성을 높이기 위한 기능으로 4.5.3절에서 자세히 설명한다.

목 차

부록 목차

요 약

클라우드나 기업 인프라에서 대규모 애플리케이션 시스템을 설계하고 배포하는데 마이크로서비스 패러다임을 사용하는 사례가 증가하고 있다. 마이크로서비스 패러다임을 사용한 애플리케이션 시스템은 상대적으로 작고, 느슨하게 결합되는 엔티티(entity)나 컴포넌트(component)로 구성된다. 이 엔티티나 컴포넌트를 마이크로서비스라고 부르며, 경량(lightweight) 통신 프로토콜을 사용하여 서로 통신한다.

마이크로서비스 기반 애플리케이션 시스템을 설계하고 배포하는 경우, 다음과 같은 이점이 있다. (a) 상대적으로 코드의 양이 적고 덜 복잡하기 때문에 개발 속도가 빠르다. 각 코드는 일반적으로 한 가지 비즈니스 기능을 구현하기 때문이다. (b) 마이크로서비스는 느슨하게 결합되기 때문에 개발팀 사이의 독립성이 보장된다. (c) 인증, 접근 제어, 서비스 검색, 통신, 부하 분산과 같은 인프라 서비스를 제공하는 배포 도구를 사용할 수 있다.

마이크로서비스의 활성화를 지원하는 일부 기술(예 : 오케스트레이션)에도 불구하고, 마이크로서비스 기반 애플리케이션의 개발과 배포에는 해결해야 할 과제가 많이 남아있다. 네트워크 보안, 신뢰성, 지연 시간은 중요한 요소이다. 마이크로서비스를 이용하여 구현된 모든 트랜잭션은 네트워크를 통해 메시지 형태로 전송되기 때문이다. 또한, 마이크로서비스가 증가할수록 공격 표면은 넓어지게 된다.

이 문서의 목적은 마이크로서비스 기반 애플리케이션을 안전하게 배포하기 위한 전략을 수립하는 것이다. 이를 위해 핵심 기능에 대한 구현 옵션을 분석하고, API 게이트웨이나 서비스 메시(service mesh)와 같은 아키텍처 프레임워크의 구성을 검토하며, 마이크로서비스에 대한 위협에 대응하기 위한 방안을 마련할 것이다.

1. 서론

마이크로서비스 패러다임을 채택한 애플리케이션 시스템이 개발되고 배포되는 사례가 증가하고 있다. 이는 신속성, 유연성, 확장성 및 기본적인 프로세스를 자동화하기 위한 도구의 가용성과 같은 장점 때문이다. 그러나, 마이크로서비스 기반 애플리케이션은 컴포넌트 간의 다양한 상호작용 방식으로 구성된 복잡한 네트워크와 결합되기 때문에 컴포넌트가 급격히 증가하는 경우, 몇몇 핵심적인 인프라 기능이 필요하다. 핵심적인 인프라 기능은 단독으로 구현되거나, API(Application Programming Interface) 게이트웨이 및 서비스 메시와 같은 아키텍처 프레임워크에 번들 또는 패키지로 제공될 수 있다. 이 문서의 목적은 핵심 기능에 대한 구현 옵션, 아키텍처 프레임워크의 구성 옵션과 마이크로서비스의 고유 위협에 대한 대책을 분석하며, 보안 전략의 아우트라인(outline)을 수립하는 것이다.

1.1. 범위

이 문서에서는 마이크로서비스 기반 애플리케이션 시스템의 배포에 사용되는 다양한 도구에 대해 논의하지 않을 것이다. 핵심 기능과 아키텍처 프레임워크에 대해서는 안전한 구현과 관련된 강조 사항을 위주로 설명할 것이다. 핵심 논점은 다음에 나열하는 세 가지 기본적인 단계를 통해 마이크로서비스 기반 애플리케이션에 대한 보안 전략을 개발하는 방법론에 대한 것이다.

- 마이크로서비스 기반 애플리케이션 시스템의 기술 연구 - 설계 원칙, 기본적인 구성 요소 및 관련 인프라 위주
- 마이크로서비스 운영 환경과 관련된 위협에 대한 집중 검토
- 보안 전략 개발을 위해 아키텍처 프레임워크(API 게이트웨이, 서비스 메시)와 관련된 최신 핵심 기능의 구현 옵션과 마이크로서비스 고유 위협에 대한 대책의 분석

1.2. 대상 독자

이 문서에서 다루고 있는 보안 전략은 다음과 같은 독자에게 유용할 것이다.

- 조직의 인프라를 마이크로서비스 아키텍처 기반의 분산 시스템으로 전환하려는 기업이나 공공기관의 IT부문에서 근무하는 CSO(Chief Security Officer), CTO(Chief Technology Officer)
- 마이크로서비스 기반 애플리케이션 시스템을 설계하려는 애플리케이션 설계 담당자

1.3. 다른 NIST 가이드라인과의 관계

이 가이드라인은 특정 아키텍처에 기반한 애플리케이션에 초점을 맞추고 있다. 그러나, 필수적인 아키텍처 컴포넌트(마이크로서비스)는 컨테이너에서 구현될 수 있다. 따라서, 아래에 나열하는 애플리케이션 컨테이너 기술과 관련한 보안 가이드라인과 권고사항은 이 문서에서 다루고 있는 애플리케이션 아키텍처의 보안 전략과 관련이 있을 수 있다.

- NIST SP 800-190, 애플리케이션 컨테이너 보안 가이드라인
- NIST IR 8176, 리눅스 애플리케이션 컨테이너 배치 관련 보안 보증 요구사항

1.4. 방법론 및 조직

마이크로서비스 기반 애플리케이션 시스템은 서버 가상화, 컨테이너, 클라우드 미들웨어와 같은 다양한 기술을 포함한다. 따라서, 마이크로서비스 기반 애플리케이션의 핵심 기능과 핵심 기능을 번들 또는 패키지로 제공하는 아키텍처 프레임워크에 주목해야 한다. 위협 분석은 마이크로서비스 기반 애플리케이션 시스템의 전체 배치 스택과 핵심 기능이 위치하는 레이어에 대해 거시적 관점으로 접근한다. 보안 전략을 개발하기 위한 전반적인 접근법은 핵심 기능에 대한 위협을 식별하고, 핵심 기능의 다양한 구현 및 아키텍처 프레임워크를 분석하며, 구현 옵션이 마이크로서비스에 특정적인 위협에 대응하도록 보장하는 것이다. 이 방법론에서 사용한 자료에 대한 로드맵은 다음과 같다.

- 마이크로서비스 인프라를 구성하는 모든 최신 핵심 기능에 대한 검토 : 2.6절
- 배치 스택의 각 레이어, 핵심 기능의 위치, 마이크로서비스에 특정적인 위협의 식별 검토 : 3장
- 핵심 기능에 대한 모든 구현 옵션 및 구현 옵션에 기반한 보안 전략 아우트라인 분석 : 4장
- 일부 핵심 기능을 단일 제품으로 제공하는 모든 아키텍처 프레임워크 및 아키텍처 프레임워크에 대한 구성 옵션에 기반한 보안 전략 아우트라인 검토 : 5장

각 장에서 설명하고 있는 내용은 다음과 같다.

- 2장 : 마이크로서비스 기반 애플리케이션 시스템에 대한 개요(개념, 설계 원칙, 비즈니스 동인, 구성 요소, 컴포넌트 상호작용 스타일, 최신 핵심 기능, 아키텍처 프레임워크)
- 3장 : 스택 관점의 위협에 대한 배경지식, 마이크로서비스 환경에 특정적인 위협
- 4장 : 마이크로서비스 기반 애플리케이션을 지원하기 위한 다양한 최신 핵심 기능과 관련된 분석, 구현 옵션 분석을 기반으로 핵심 기능을 구현하기 위한 보안 전략
- 5장 : 마이크로서비스 기반 애플리케이션 인프라에서 필요한 핵심 기능을 번들로 제공하는 아키텍처 프레임워크 분석, 아키텍처 프레임워크를 구성하기 위한 보안 전략

2. 배경지식

이 장에서는 마이크로서비스 기반 애플리케이션 시스템을 개발하고 배포하는데 사용되는 기술에 대해 설명한다. 이를 위해 기본적인 설계 원칙, 구성 요소, 다양한 아키텍처 옵션을 사용하기 위해 구성 요소를 구성할 수 있는 다양한 방법을 언급한다. 기술에 대한 포괄적인 설명보다, 마이크로서비스 기반 애플리케이션 시스템에 대한 보안 위협을 식별하고 안전한 구현 전략을 수립하기 위해 컴포넌트와 개념에 대해 충분히 설명한다.

2.1. 마이크로서비스의 개념

마이크로서비스 기반 애플리케이션 시스템은 다수의 컴포넌트(마이크로서비스)로 구성된다. 각 마이크로서비스는 동기 RPC(Remote Procedure Call, 원격 프로시저 호출) 또는 비동기 메시지 시스템을 통해 서로 통신한다. 각 마이크로서비스는 서로 다른 비즈니스 프로세스나 기능(예 : 고객 정보 저장, 제품 목록 저장 및 표시, 고객 주문 처리 등) 하나를 구현하는 것이 일반적이다. 다수의 비즈니스 프로세스나 기능을 구현하는 것은 흔하지 않다. 마이크로서비스는 데이터베이스에 접근하거나 메시지를 보내는 것과 같은 기능을 수행하기 위해 자체 비즈니스 로직과 다양한 어댑터를 보유한 작은 애플리케이션이다. 일부 마이크로서비스는 REST[1]ful API를 공개[1]하고, 다른 마이크로서비스이나 애플리케이션의 클라이언트가 RESTful API를 이용[2]한다. 다른 마이크로서비스는 웹 UI를 구현할 수 있다. 마이크로서비스 인스턴스는 애플리케이션 서버, VM(Virtual Machine, 가상머신), 컨테이너에서 프로세스로 실행되도록 구성할 수 있다.

마이크로서비스 기반 애플리케이션은 클라우드 서비스 용도가 아니라, 순수하게 엔터프라이즈 애플리케이션 용도로 구현할 수도 있다. 하지만, 마이크로서비스 기반 애플리케이션은 서비스 기반 아키텍처, API 주도의 통신, 컨테이너 기반 인프라, DevOps 프로세스[2] 기반의 클라우드 네이티브 애플리케이션으로 개발하는 것이 일반적이다. 온프레미스(on-premise)의 경우, API 기반 통신을 사용하여 느슨하게 결합된 서비스 기반 아키텍처보다는 긴밀하게 통합된 애플리케이션 모듈을 사용하는 서버 중심의 인프라에서 개발되고 배포되기 때문이다.

---

1 Representational State Transfer

2 DevOps 프로세스는 개발자, 품질 보증팀, 보안 전문가, IT 운영자 및 업무 이해관계자 사이에서 지속적인 개선, 빠른 개발, 지속적인 배포 및 개발 협업을 말한다.[3]

2.2. 마이크로서비스의 설계 원칙

마이크로서비스는 다음과 같은 동인(driver)을 기반으로 설계한다.[4]

- 각 마이크로서비스는 다른 마이크로서비스와 독립적으로 관리되고, 복제되고, 확장되고, 업그레이드되고, 배치되어야 한다.
- 각 마이크로서비스는 단일 기능을 보유해야 하며, 독립적으로 서비스되어도 문제가 발생하지 않는 범위(Bounded Context)에서 동작해야 한다. 특히, 다른 서비스에 대한 책임과 의존을 제한해야 한다.
- 모든 마이크로서비스는 장애 및 복구를 고려하여 설계해야 한다. 따라서, 가능한 한 스테이트리스(stateless)해야 한다.
- 상태 관리를 위해 기존에 존재하는 신뢰할 수 있는 서비스(예 : 데이터베이스, 캐시, 디렉터리)를 사용해야 한다.

상기 동인으로부터 마이크로서비스에 대한 다음과 같은 설계 원칙이 도출된다.

- 자율성(Autonomy)
- 느슨한 결합(Loose coupling)
- 재사용성(Re-usability)
- 조립성(Composability)
- 고장 허용 한계(Fault tolerance)
- 검색 용이성(Discoverability)
- 비즈니스 프로세스에 대응하는 API(APIs alignment with business processes)

2.3. 비즈니스 동인

마이크로서비스 기반 애플리케이션 도입을 추진하게 하는 비즈니스 동인은 이 문서의 주제와 거의 관련이 없다. 그러나, 사용자 및 조직의 행위 관점에서 관련이 있는 비즈니스 동인을 식별하고 정리하는 것은 유용하다.[5]

- 유비쿼터스 접근(Ubiquitous access) : 사용자는 다양한 기기(예 : 웹 브라우저, 모바일 기기)에서 애플리케이션에 접근하기를 원한다.
- 확장성(Scalability) : 애플리케이션은 사용자나 사용률이 증가하더라도 가용성을 유지할 수 있도록 확장성이 높아야 한다.
- 빠른 개발(Agile development) : 조직은 조직의 변화(프로세스 및 구조)와 시장의 요구에 신속하게 대응하기 위해 빠른 업데이트를 원한다.

2.4. 구성 요소

특정 기술에 종속적이지 않으며 가벼운 메커니즘으로 서로 통신하는 작고 독립적인 엔티티(엔드포인트)로 구성된 아키텍처 스타일이나 디자인 패턴을 사용하여 마이크로서비스 기반 애플리케이션(예 : 분산된 엔터프라이즈 또는 웹 애플리케이션[1])을 구축한다. 이러한 엔드포인트는 잘 정의된 API를 이용하여 구현된다. SOAP(Simple Object Access Protocol) 또는 REST(HTTP 프로토콜)와 같은 몇 가지 엔드포인트 API 유형이 있다. 각 엔드포인트는 서로 다른 비즈니스 기능을 제공하며, 자체 데이터 스토어나 저장소를 가질 것이다. 비즈니스 기능을 서비스라고 부른다. 웹 브라우저나 모바일 기기와 같이 다양한 플랫폼이나 클라이언트 유형에서 클라이언트라고 부르는 컴포넌트를 사용하여 서비스에 접근한다. 서비스와 클라이언트가 함께, 완전한 마이크로서비스 기반 애플리케이션 시스템을 형성한다. 마이크로서비스 기반 애플리케이션 시스템의 서비스는 다음과 같이 구분할 수 있다.

- 애플리케이션 기능 서비스
- 인프라 서비스 : 단독으로 구현되거나, 아키텍처 프레임워크(예 : API 게이트웨이, 서비스 메시)에서 번들로 제공된다. 인프라 서비스에는 인증과 인가, 서비스 등록 및 검색, 보안 모니터링이 포함된다. 이 문서에서는 인프라 서비스를 핵심 기능(core feature)으로 부른다.

마이크로서비스 기반 애플리케이션 시스템에서 각 서비스는 다른 기술을 사용하여 구축될 수 있다. 이러한 사실에서 기술적 이질성(technical heterogeneity)이라는 개념이 부각된다. 기술적 이질성이란, 마이크로서비스 기반 애플리케이션 시스템에서 각 서비스가 다른 프로그래밍 언어, 개발 플랫폼, 다른 데이터 스토리지 기술을 이용하여 작성될 수 있음을 의미한다. 기술적 이질성은 개발자가 서비스 개발에 적절한 도구와 언어를 선택할 수 있게 해준다. 따라서, 마이크로서비스 기반 애플리케이션 시스템을 구성하는 서비스는 다른 개발 언어(예 : 루비, Go, 자바)로 구축될 수 있고, 다른 데이터베이스(예 : 도큐먼트 데이터스토어[1], 그래픽 DB, 멀티미디어 DB)를 호스팅할 수 있다. 한 개의 서비스는 한 개의 팀(마이크로서비스 또는 DevOps 팀)이 개발한다. 서비스 기능이나 서비스 계약에 대한 합의가 이루어지면, 각 팀은 담당 서비스에 대해 개발 및 운영과 관련한 모든 요구사항을 구현한다. 이 때, 개발 및 배포 기술에 대해서는 높은 자율성이 주어진다.[6]

서비스는 각각 다른 노드에 배치된다. 각 서비스는 원격 호출을 사용하여 서로 통신한다. 이러한 방식은 네트워크의 지연 시간이 높은 경우, 시스템 성능에 영향을 미칠 수 있다. 따라서, 경량 통신 인프라가 필요하다.

1 도큐먼트 데이터스토어(document datastore)는 NoSQL 데이터베이스의 한 분류로 MongoDB, CouchDB 등이 있다. 도큐먼트 지향 데이터베이스(document-oriented database)라는 용어를 더 많이 사용한다.

특정 서비스가 CPU(Central Processing Unit) 및 메모리가 충분하지 않아 성능에 병목 현상이 발생하는 경우, 해당 서비스만 선택적으로 확장할 수 있다. 그 외 다른 서비스는 성능이 낮고 저렴한 하드웨어를 계속 사용할 수 있다. 서비스의 기능은 목적에 따라 다른 방법으로 사용된다. 따라서, 재사용성과 조립성이 부각된다. 한 가지 예로 고객 데이터베이스 서비스의 경우, 발송 부서에서는 배송 확인 서류(bills of lading)를 준비하기 위해 사용하며, 회계 부서에서는 청구서를 보내기 위해 사용한다.

2.5. 마이크로서비스의 상호작용

모놀리식(monolithc) 애플리케이션에서, 각 컴포넌트(프로시저 또는 함수)는 코드 내에서 메소드나 함수를 호출함으로써 다른 컴포넌트를 실행한다. 마이크로서비스 기반 애플리케이션에서 각 서비스는 일반적으로 각각 다른 네트워크 노드에서 실행되는 프로세스이며, IPC(Inter-Process Communication)를 사용하여 다른 서비스와 통신한다.[7] 추가적으로, 서비스는 IDL(Interface Definition Language, 인터페이스 정의 언어)를 이용하여 정의되며, 산출물로 API(Application Programming Interface)가 생성된다. 서비스 개발의 첫 단계에는 인터페이스를 정의하는 과정이 포함되며, 클라이언트 개발자와 함께 수 차례에 걸쳐 반복적으로 검토한 후 서비스 구현을 시작한다.

IPC 메커니즘의 선택은 API의 특성을 결정한다.[7] 다음 표는 각 IPC 메커니즘에 따른 API의 특성을 보여준다.

IPC 메커니즘	API의 특성
비동기식, 메시지 기반(예 : AMQP[1], STOMP[2])	메시지 채널과 메시지 유형으로 구성
동기식 요청/응답(예 : HTTP 기반 REST, Thrift)	URL[3]과 요청/응답 포맷으로 구성

표 1 – IPC 메커니즘과 API 유형

IPC 통신에서 사용하는 메시지 포맷은 다양한 유형을 사용할 수 있다. JSON(JavaScript Object Notation)이나 XML(Extensible Markup Language)과 같이 사람이 읽을 수 있는 텍스트 기반 포맷, 아파치 Avro나 프로토콜 버퍼와 같이 기계만 읽을 수 있는 바이너리 포맷이 있다.

[1] Advanced Message Queuing Protocol

[2] Simple (or Streaming) Text Oriented Messaging Protocol

[3] Uniform Resource Locator

앞에서 언급한 자율성의 원칙에 따르면, 각 마이크로서비스는 애플리케이션 스택의 모든 기능을 제공하는 독립적인 엔티티여야 할 필요가 있다. 그러나, 다양한 비즈니스 프로세스 기능을 제공하는 마이크로서비스 기반 애플리케이션의 경우, 마이크로서비스는 몇 가지 방식에서 항상 다른 마이크로서비스에 종속적(예 : 데이터)이다. 주문, 배송, 청구와 같은 기능을 제공하는 온라인 쇼핑 애플리케이션의 경우, 배송 마이크로서비스는 배송 기록을 생성하기 위해 주문 마이크로서비스의 아직 완료되지 않은 주문 데이터에 종속된다. 따라서, 항상 자율성을 유지하면서 마이크로서비스를 결합할 필요가 있다. 결합을 형성하는 방법은 비즈니스 프로세스와 IT 인프라에 의해 주로 결정되며, 상호작용 패턴, 메시지 패턴, 서비스 사용 방식(consumption mode) 등 다양한 접근법이 존재한다. 이 문서에서는 상호작용 패턴이라는 용어를 사용하며, 주요 상호작용 패턴은 다음과 같다.

요청-응답 : 요청은 질의(정보 검색)와 명령(비즈니스 기능의 상태 변경)으로 구분된다.[2] 질의의 경우, 마이크로서비스는 정보를 요청하거나, 어떤 행동을 취하도록 요청하고, 기능적으로 응답을 기다린다. 정보를 요청하는 목적은 정보를 제공하기 위한 검색에 있다. 명령의 경우, 한 마이크로서비스가 다른 마이크로서비스에게 어떤 행동을 취하기를 요청한다. 어떤 행동이란, 비즈니스 기능의 상태를 변경(예 : 고객이 프로필을 변경하거나, 주문을 제출)시키는 것 등이다. 요청-응답 패턴에서는 두 마이크로서비스 사이의 강력한 런타임 종속성이 존재한다. 런타임 종송석은 다음의 두 가지 형태로 나타난다.

- 마이크로서비스는 다른 마이크로서비스가 이용 가능한 경우에만 그 기능을 실행할 수 있다.
- 마이크로서비스는 다른 마이크로서비스에게 요청이 성공적으로 전달됨을 보장해야 한다.

요청-응답 프로토콜의 통신 특성 때문에, HTTP와 같은 동기식 통신 프로토콜을 사용한다. 마이크로서비스가 REST API로 구현된다면, 마이크로서비스 사이의 메시지는 HTTP REST API 호출이 된다. REST API는 마이크로서비스 인터페이스 정의와 발행(publication)을 위해 개발된 표준화된 언어(예 : RAML, RESTful API Modeling Language)를 사용하여 정의하는 경우가 많다. HTTP는 통신의 블로킹(blocking) 유형으로, 요청을 시작한 클라이언트는 응답을 받은 후에만 작업을 계속할 수 있다.

발행-구독 : 복잡한 비즈니스 프로세스나 트랜잭션을 구현하기 위해 다수의 마이크로서비스가 협업해야 하는 경우, 발행-구독 패턴을 사용한다. 발행-구독 패턴은 비즈니스 도메인 이벤트 주도 접근법(business domain event-driven approach)이나 도메인 이벤트 구독 접근법(domain event subscription approach)이라고도 부른다. 발행-구독 패턴에서, 마이크로서비스는 자신을 비즈니스 도메인 이벤트에 등록하거나, 비즈니스 도메인 이벤트(예 : 관심이 있는 특정 정보, 특정 요청의 처리)를 구독한다. 비즈니스 도메인 이벤트는 이벤트 버스 인터페이스를 통해 메시지 브로커에게 발행된다. 마이크로서비스는 이벤트 주도 API를 사용하여 구축되며, MQTT(Message Queuing Telemetry Transport), AMQP(Advanced Message Queuing Protocol), 카프카(Kafka) 메시지와 같은 비동기 메시지 프로토콜을 사용한다. 비동기 메시지 프로토콜은 알림과 구독을 지원한다. 비동기 프로토콜에서, 메시지를 보내는 측은 일반적으로 응답을 기다리지 않고, 메시지를 메시지 에이전트(예 : RabbitMQ 큐)로 보낼 뿐이다. 발행-구독 패턴을 사용한 사례로 특정한 이벤트에 따라 다수의 마이크로서비스에 데이터를 업데이트하도록 전파하는 것을 꼽을 수 있다.[8]

2.6. 마이크로서비스의 최신 핵심 기능

마이크로서비스 기반 애플리케이션 환경에서는 통신 인프라가 중요하다. 따라서, 많은 배포 상황에서 다양한 핵심 기능이 필요하다. 이미 언급한 바와 같이, 핵심 기능은 독립적으로 구현되거나, API 게이트웨이나 서비스 메시와 같은 아키텍처 프레임워크에서 번들로 제공될 수 있다. API 게이트웨이 내에서도 핵심 기능은 서비스 조합으로 구현되거나, 코드 내부에 직접 구현될 수 있다. 이러한 기능에는 인증, 접근 제어, 서비스 검색, 부하 분산, 캐시(cache), 애플리케이션 헬스 체크(health check) 및 모니터링[2]이 포함되며, 이외 다른 기능도 포함될 수 있다. 아래에 이러한 기능[5]에 대해 간략히 설명한다.

- 인증과 접근 제어 – 인증과 접근 정책은 마이크로서비스가 노출하는 API의 유형에 따라 다르다. API의 유형에는 퍼블릭 API, 프라이빗 API, 파트너 API(비즈니스 파트너만 사용 가능)가 있다.

- 서비스 검색 – 레거시 분산 시스템은 다수의 서비스가 지정된 IP 주소와 포트 번호로 동작하도록 구성된다. 마이크로서비스 기반 애플리케이션에서는 다음과 같은 시나리오가 존재하며, 강력한 서비스 검색 메커니즘이 필요하다.
  (a) IP 주소와 포트 번호가 동적으로 변경되고, 각 서비스와 관계되는 서비스 또는 인스턴스의 수가 상당하다.
  (b) 각 마이크로서비스는 VM(Virtual Machine) 또는 컨테이너에서 구현될 수 있다. 이 경우, 특히 IAAS(Infrastructure as a Service) 또는 SAAS(Software as a Service)와 같은 클라우드 서비스를 이용할 경우 IP 주소가 동적으로 할당될 수 있다.
  (c) 자동 확장(autoscaling)과 같은 기능을 사용할 경우, 서비스와 관계된 인스턴스의 수는 부하의 변동에 따라 달라질 수 있다.

- 보안 모니터링 및 분석 – 공격을 탐지하고, 가용성에 영향을 미칠 수 있는 서비스 저하 요인을 식별하기 위해 마이크로서비스로 유입/유출되는 네트워크 트래픽을 분석 기능으로 모니터링해야 한다. 추가적으로 일반적인 로그인 기능도 모니터링해야 한다.

다음의 핵심 기능을 구현하기 위해서는 일반적으로 API 게이트웨이나 마이크로 게이트웨이가 필요하다.

- 엔드포인트 최적화 – 여기에는 몇몇 기능이 포함된다.
  (a) 요청 및 응답 폐쇄 : 대부분의 비즈니스 트랜잭션은 미리 정의된 순서대로 다수의 마이크로서비스를 호출한다. API 게이트웨이는 이러한 상황을 단순하게 할 수 있다. API 게이트웨이는 클라이언트에게 엔드포인트로 노출된다. API 게이트웨이는 각 마이크로서비스를 호출하는데 필요한 다수의 요청을 자동으로 생성한다. 또한, 각 마이크로서비스의 응답을 통합하여, 단 하나의 응답을 클라이언트에게 보낸다.

(b) API 변환(Transformation) : API 게이트웨이는 클라이언트에게 퍼블릭 인터페이스를 제공할 수 있다. 퍼블릭 인터페이스는 특정한 요청에 맞추기 위해 반드시 호출해야 하는 개인(individual) API와 다르다. 이러한 기능을 API 변환이라고 하며, 다음과 같은 것을 가능하게 한다.

(ⅰ) 개별 마이크로서비스의 구현을 변경하거나, 심지어 API도 변경할 수 있게 한다.

(ⅱ) 기존 모노리식 애플리케이션을 마이크로서비스 기반 애플리케이션으로 전환하는 과정(모놀리식 애플리케이션의 지속적인 분할 → 백그라운드에서 마이크로서비스 API 생성 → API 변환 설정 변경)에서도 클라이언트가 API 게이트웨이를 통해 지속적으로 서비스에 접근할 수 있게 한다.

(c) 프로토콜 변환(Translation) : 클라이언트는 마이크로서비스 엔드포인트와 HTTPS(Hypertext Transfer Protocol Secure) 같은 웹 프로토콜로 통신하면서, 마이크로서비스 간의 통신에는 RPC/Thrift 또는 AMQP 같은 동기 프로토콜을 사용될 수 있다. 클라이언트의 요청을 처리하는 데 필요한 프로토콜 변환은 일반적으로 API 게이트웨이가 수행한다.

- 서킷브레이커 – 마이크로서비스 인스턴스의 요청 실패에 대한 임계값을 설정하고, 임계값을 초과하는 인스턴스로 요청이 전달되지 않도록 하는 기능이다. 서킷브레이커는 실패가 연속되지 않도록 하고, 해당 인스턴스를 복구(로그 분석-필요 수정사항 구현-업데이트)하기 위한 시간을 확보해 준다.

- 부하 분산 : 동일한 서비스에 대해 다수의 인스턴스가 필요할 수 있다. 과부하로 인한 응답 지연이나 서비스 중단을 방지하기 위해 인스턴스 별로 부하를 고르게 분산해야 한다.

- 사용량 제한(스로틀링) : 모든 클라이언트에게 서비스의 지속적인 가용성을 보장하기 위해 마이크로서비스로 유입되는 요청을 제한하여야 한다.

- 블루/그린 배포(blue/green deployment) : 새로운 버전의 마이크로서비스가 배포된 경우, 이전 버전에 대한 요청을 새로운 버전으로 리다이렉트할 수 있다. API 게이트웨이가 신-구 버전의 위치를 인식하도록 프로그래밍할 수 있기 때문이다.

- 카나리 릴리즈(canary release) – 모든 운영 시나리오에 대해 응답의 정확성이나 성능을 완전히 알 수 없기 때문에 초기에는 제한된 트래픽만 새로운 버전의 마이크로서비스로 보낸다. 운영 특성과 관련하여 충분한 데이터가 수집되면, 모든 요청을 새로운 버전의 마이크로서비스로 보낼 수 있다.

2.7. 마이크로서비스의 아키텍처 프레임워크

마이크로서비스 기반 애플리케이션에서 통신의 신뢰성/복원성/안전성을 보장하는 핵심 기능을 번들 또는 패키지로 제공하는 주요 아키텍처 프레임워크에는 다음 두 가지가 있다.

- API 게이트웨이(마이크로 게이트웨이의 유/무와 관련 없음)
- 서비스 메시

마이크로서비스 기반 애플리케이션 시스템의 운영 환경에서 이러한 프레임워크의 역할은 다음과 같다.[4]

아키텍처 프레임워크	전체 아키텍처에서의 역할
API 게이트웨이 (마이크로 게이트웨이의 유/무와 관련 없음)	클라이언트-서버간, 서버-서버간 트래픽 제어를 위해 사용한다.
서비스 메시	마이크로서비스가 컨테이너를 사용하여 구현할 때, 오직 서버-서버간 트래픽을 위해 배치한다. 마이크로서비스가 VM 또는 애플리케이션 서버에 위치한 경우에도 사용할 수 있다.

표 2 - 마이크로서비스 운영에선 아키텍처 프레임워크의 역할

API 게이트웨이와 서비스 메시의 대표적인 기능은 서비스 검색, 부하 분산, 고장 탐지/대응 및 공격 모니터링이 포함된다.

2.7.1. API 게이트웨이

API 게이트웨이는 마이크로서비스 기반 애플리케이션 시스템에서 인기있는 아키텍처 프레임워크이다. 엔드포인트가 단 하나의 서버일 수 있는 모놀로식 애플리케이션과 다르게, 마이크로서비스 기반 애플리케이션은 다수의 세분화된 엔드포인트로 구성된다. 따라서, 모든 클라이언트에게 다수의 마이크로서비스로 구성된 애플리케이션에 대해 하나의 진입점을 제공하는 것이 좋다. API 게이트웨이를 배치하는 또 다른 상황은 조직이 모놀리식 엔터프라이즈 애플리케이션의 컴포넌트를 독립적인 마이크로서비스로 대체하면서 점진적으로 마이그레이션(migration)할 때 백엔드의 프론트엔드 역할을 수행하는 경우이다. 클라이언트가 다수의 엔드포인트와 직접 통신하는 경우, P2P 연결(point-to-point connection)이 너무 많아진다.

API 게이트웨이의 주요 기능은 인바운드(inbound) 요청을 다운스트림(down-stream) 서비스로 정확하게 라우팅하는 것이다. 또한, 부가적으로 프로토콜 변환[1]을 수행하고, 요청을 작성하기도 한다. 드물지만 요청은 BFF(Backend for Frontend)의 일부로 사용되기 때문에, 폼 팩터(예 : 브라우저, 모바일 기기)가 다른 클라이언트를 지원해야 한다. 클라이언트의 모든 요청은 일단 API 게이트웨이를 통과한다. 그 후 API 게이트웨이는 적절한 마이크로서비스로 요청을 라우팅한다. API 게이트웨이는 다수의 마이크로서비스를 호출하고 그 결과를 종합하여 응답한다.

API 게이트웨이를 통해 접속할 수 있는 다수의 API 및 마이크로서비스를 게이트웨이(예 : 모바일 API 또는 모바일 서비스)의 입력 포트를 정의함으로써 지정하거나, 요청 파라미터(요청된 서비스 이름을 포함)를 갖는 API 게이트웨이 서비스를 배포함으로써 동적으로 지정할 수 있다.[9] 따라서, API 게이트웨이(클라이언트와 마이크로서비스 사이에 위치)는 프락시에서 여러 서비스를 종합하는 패턴을 나타낸다. 많은 API 게이트웨이가 Jolie, 자바스크립트, 자바와 같이 다른 개발 언어로 작성된 API를 지원할 수 있도록 구현된다.

API 게이트웨이는 마이크로서비스의 진입점이기 때문에 주요 서비스인 요청 제어 이외에도 서비스 검색, 인증, 접근 제어, 부하 분산, 캐싱, 클라이언트 유형별 커스텀 API 제공, 애플리케이션 헬스 체크, 서비스 모니터링, 공격 탐지, 공격 대응, 보안 로깅/모니터링, 서킷브레이커와 같은 인프라 서비스가 탑재되어야 한다. 이러한 추가적인 기능은 API 게이트웨이에 두 가지 방법으로 구현된다.

- 각 기능(예 : 서비스 검색을 위한 서비스 레지스트리)이 개발된 특정 서비스를 조립
- API 게이트웨이가 사용하는 코드 내부에 기능을 직접 구현

1 프로토콜 변환은 웹 프로토콜(HTTP 및 웹소켓 등)과 내부적으로 사용되는 웹 비친화적 프로토콜(AMQP, Thrift 바이너리 RPC 등) 간의 변환을 말한다.

게이트웨이 구현

클라이언트 유형별로 요청을 제어하는 게이트웨이 로직이 과도하게 많아지는 것을 방지하기 위해 게이트웨이를 다수로 분리한다.[8] 이러한 패턴을 BEF(backends for frontends, 프론트엔드에 대한 백엔드)라고 부른다. BEF에서는 클라이언트 유형별로 서비스 요청을 수집하기 위한 게이트웨이(예 : 웹 앱 BEF, 모바일 앱 BEF)가 주어진다. 각 백엔드(게이트웨이)는 해당 프론트엔드(클라이언트)와 밀접하게 연계되며, 일반적으로 동일한 팀에서 개발한다. BEF에서 제공하는 기능은 GraphQL로도 제공할 수 있다. GraphQL은 클라이언트가 요청할 때, 응답의 데이터 형태를 지정할 수 있다.

마이크로서비스 기반 애플리케이션의 API 관리는 모놀로식 API 게이트웨이 아키텍처 또는 분산된 API 게이트웨이 아키텍처를 통해 구현될 수 있다. 모놀로식 API 게이트웨이 아키텍처에는 API 게이트웨이가 하나만 존재한다. API 게이트웨이는 일반적으로 엔터프라이즈 네트워크의 경계(예 : DMZ[1])에 배치되고, 엔터프라이즈 수준에서 API로 모든 서비스를 제공한다. 분산 API 게이트웨이 아키텍처에는 마이크로 게이트웨이의 인스턴스가 다수 존재한다. 마이크로 게이트웨이는 마이크로서비스 API에 더 가깝게 배치된다.[10] 마이크로 게이트웨이는 일반적으로 용량이 작고 스크립트가 가능한 API 게이트웨이로서, 정책을 정의하고 시행하는데 사용할 수 있다. 따라서, 마이크로 게이트웨이는 서비스별로 고유한 보안 정책을 통해 보호되어야 하는 마이크로서비스 기반 애플리케이션에 적합하다.

마이크로 게이트웨이는 일반적으로 Node.js와 같은 개발 플랫폼을 사용하여 독립적인 컨테이너로 구현된다. 이것은 서비스 메시 아키텍처의 사이드카 프록시(sidecar proxy)와는 다르다. 사이드카 프록시는 API 엔드포인트 자체에 구현된다. 사이드카 프록시에 대해서는 2.7.2절에서 설명한다. 보안 정책을 인코딩하고 게이트웨이에 입력할 수 있는 방법은 다양하다. 한 가지 방법은 정책을 JSON 포맷으로 인코딩하고, 그래픽 정책 관리 인터페이스를 통해 입력하는 것이다. 마이크로 게이트웨이에는 애플리케이션 요청과 응답에 대한 정책이 모두 입력되어야 한다. 정책과 정책 적용이 컨테이너로 구현된 경우, 정책은 변경할 수 없다. 따라서, 보안 위반이나 충돌을 유발할 수 있는 우발적이고 의도하지 않은 변경을 어느 정도 예방할 수 있다. 즉, 마이크로 게이트웨이를 컨테이너로 구현했을 때, 이러한 유형의 변경을 방지한다. 보안 정책을 업데이트하기 위해서는 마이크로 게이트웨이를 재배치해야 하기 때문이다. 마이크로 게이트웨이(모든 마이크로서비스 인스턴스별로 배치)는 보호하도록 설계된 마이크로서비스의 작동 상태를 추적하기 위해 서비스 등록 모듈과 모니터링 모듈과 통신해야 한다.

[1] Demilitarized Zone

2.7.2. 서비스 메시

서비스 메시는 서비스 검색, 라우팅, 내부 부하 분산, 트래픽 구성, 암호화, 인증/인가, 메트릭, 모니터링을 통해 서비스간 통신을 용이하게 하는 전용 인프라 레이어이다. 서비스 메시는 네트워크 토폴로지가 변경되는 상황(서비스 인스턴스가 오프라인 상태가 되고, 지속적으로 재배치)에서 정책을 통해 네트워크 행위/노드 식별/트래픽 흐름을 선언적으로 정의할 수 있다. 서비스 메시는 OSI (Open System Interconnection) 모델에서 전송 계층(Transport Layer) 상위의 추상화 계층을 차지하며, 서비스와 세션 계층(OSI 모델의 5계층) 간의 문제를 해결할 수 있는 네트워크 모델로 간주할 수 있다.[11] 그러나, 세부적인 인가는 마이크로서비스에서 수행해야 할 필요가 있다. 마이크로서비스는 비즈니스 로직을 완전하게 인지하고 있는 유일한 엔티티이기 때문이다. 서비스 메시는 개념적으로 데이터 플레인과 컨트롤 플레인을 갖는다. 데이터 플레인은 서비스 인스턴스 사이에서 서비스 프락시를 통해 애플리케이션 요청 트래픽을 전달한다. 컨트롤 플레인은 데이터 플레인을 설정하고, 텔레메트리(telemetry, 원격 측정)를 종합하며, 부하 분산/서킷브레이커/사용량 제한과 같은 다양한 기능을 통해 네트워크 동작을 변경하기 위한 API를 제공한다.

서비스 메시는 마이크로서비스 애플리케이션을 구성하는 각 서비스 별로 작은 프록시 서버 인스턴스를 형성한다. 이렇게 전문화된 프록시 서버를 서비스 메시 용어로 사이드카 프록시(sidecar proxy)라고도 부른다.[12] 사이드카 프록시는 데이터 플레인에 구성되며, 접근 제어/통신과 관련된 보안 적용을 위해 필요한 런타임 작업은 데이터 플레인에서 사이드카 프록시에 정책(예 : 접근 제어 정책)을 주입해야 가능하다. 또한, 마이크로서비스의 코드를 수정하지 않고도 정책을 동적으로 변경할 수 있는 유연성을 제공한다.

2.8. 모놀리식 아키텍처와 비교

마이크로서비스 아키텍처와 모놀리식 아키텍처(모든 레거시 애플리케이션에서 사용)를 완전하게 비교하기 위해서는, 두 가지 아키텍처 스타일로 개발된 애플리케이션의 기능을 비교하고, 특정 비즈니스 프로세스를 두 가지 아키텍처로 구현한 애플리케이션 예제가 필요하다. 이에 대한 자세한 설명은 부록 A에 수록되어 있다.

2.9. 서비스 지향 아키텍처(SOA, Service-Oriented Architecture)와 비교

마이크로서비스 아키텍처와 서비스 지향 아키텍처는 다음과 같은 공통된 기술 개념 때문에 많은 부분에서 유사하다.[13]

- 서비스 – 애플리케이션 시스템은 서비스라고 부르는 자체적인 엔티티 또는 아티팩트를 통해 다양한 기능을 제공한다. 서비스는 다른 속성(visible, discoverable, stateless, reusable, composable)을 갖거나, 기술적인 다양성을 갖는다.
- 상호 운용성 – 한 서비스는 다른 서비스를 호출할 수 있다. 서비스 지향 아키텍처의 경우에는 ESB(Enterprise Service Bus)와 같은 아티팩트를 사용하여 호출하며, 마이크로서비스 아키텍처의 경우에는 네트워크에서 RPC(Remote Procedural Call)를 사용하여 호출한다.
- 느슨한 결합 – 한 서비스를 변경할 때, 다른 서비스를 변경할 필요가 없도록 서비스 사이의 의존성을 최소화한다.

위에서 명시한 세 가지 공통된 기술 개념에도 불구하고, 서비스 지향 아키텍처와 마이크로서비스 아키텍처의 관계에 대해서는 다음과 같은 기술적인 견해가 있다.[13]

- 마이크로서비스 아키텍처와 서비스 지향 아키텍처는 별개의 아키텍처 스타일이다.
- 마이크로서비스 아키텍처는 서비스 지향 아키텍처의 한 가지 패턴이다.
- 마이크로서비스 아키텍처는 세련된 서비스 지향 아키텍처이다.

서비스 지향 아키텍처와 마이크로서비스 아키텍처의 차이는 구체적인 구현(개발/배포 패러다임 및 기술)을 제외하면, 아키텍처 스타일과는 관련이 없다는 것이 가장 보편적인 견해이다.[2]

2.10. 마이크로서비스의 장점

- 대규모 애플리케이션의 경우, 애플리케이션을 느슨하게 결합된 컴포넌트로 분리하는 것은 각 컴포넌트를 개발하는 팀들 사이의 독립성을 허용한다. 각 팀은 개발중인 컴포넌트에 대한 적합성을 기반으로 개발 플랫폼, 개발 도구, 개발 언어, 미들웨어, 하드웨어를 선택함으로서 최적화할 수 있다.

- 각 컴포넌트를 독립적으로 확장할 수 있다. 자원을 지정하여 할당하는 것은 자원 활용을 최대화할 수 있게 해 준다.

- 컴포넌트가 HTTP RESTful 인터페이스를 가지고 있는 경우, 인터페이스가 변경되지 않는 한 애플리케이션의 전반적인 기능을 중단하지 않고 구현을 변경할 수 있다.

- 각 컴포넌트에 포함된 코드는 상대적으로 규모가 작기 때문에 개발팀이 업데이트를 빠르게 제작할 수 있다. 또한, 비즈니스 프로세스나 시장 상황의 변화에 애플리케이션이 민첩하게 대응할 수 있다.

- 컴포넌트 사이의 느슨한 결합은 마이크로서비스의 중단을 격리한다. 따라서, 애플리케이션의 다른 컴포넌트나 다른 부분에 대한 도미노 효과 없이 해당 서비스로 영향을 제한한다.

- 컴포넌트가 비동기 이벤트 처리 메커니즘을 사용하여 서로 연결되어 있을 때, 컴포넌트 중단의 영향은 일시적이다. 컴포넌트가 다시 가동되기 시작할 때 필요한 기능이 자동으로 실행되어, 비즈니스 프로세스의 전반적인 무결성이 유지되기 때문이다.

- 서비스를 비즈니스 기능에 맞추어 정의함으로써 (또는 비즈니스 프로세스/기능에 맞추어 전반적인 애플리케이션 기능을 분해함으로써) 마이크로서비스 기반 시스템의 전반적인 아키텍처를 조직의 구조에 대응시킨다. 따라서, 조직의 구조와 관련된 비즈니스 프로세스가 변경되고, 결과적으로 관련된 서비스를 수정하여 배치해야 할 때, 신속하게 대응할 수 있게 한다.

- 마이크로서비스에서 기능의 독립성은 애플리케이션 전반에서 코드를 더 많이 재사용할 수 있게 한다.

2.11. 마이크로서비스의 단점

- 단일 애플리케이션 대신, 다수 컴포넌트(마이크로서비스)를 모니터링해야 한다. 각 컴포넌트의 상태, 전체 애플리케이션의 상태를 확인하기 위해 중앙 콘솔이 필요하다. 따라서, 분산 모니터링과 중앙화된 모니터링 기능을 갖는 인프라를 조성해야 한다.

- 컴포넌트가 다수인 경우, 모든 컴포넌트는 언제든지 중단될 수 있기 때문에 가용성 문제가 발생한다.

- 한 컴포넌트에서 다른 컴포넌트를 호출할 때, 일부 클라이언트는 최신 버전을 호출하고, 일부 클라이언트는 이전 버전을 호출해야 할 수 있다. (예 : 버전 관리)

- 통합 테스트는 모든 컴포넌트가 작동하고 서로 통신해야 하는 테스트 환경이 필요하기 때문에 매우 어렵다

- 마이크로서비스 기반 애플리케이션 내부의 상호작용을 API 호출로 설계할 때, API를 안전하게 관리하기 위해 필요한 모든 프로세스를 구현해야 한다.

- 마이크로서비스 아키텍처는 심층 방어(defense in depth)를 와해시킬 수 있다. 다수 아키텍처가 웹 서버, 백엔드 서비스, 데이터베이스를 갖는다. 백엔드 서비스는 노출된 웹 서버와 데이터베이스(민감한 데이터) 사이에서 보호 계층(hardened layer)으로 동작할 수 있다. 마이크로서비스 아키텍처는 이를 와해하는 경향이 있다. 웹 서버와 백엔드 서비스는 마이크로서비스로 분해된다. 마이크로서비스는 이전의 모델보다 잠재적으로 더 많이 노출되기 때문에, 호출자(caller)와 민감한 데이터 사이의 보호 계층(protection layer)이 축소될 수 있다. 따라서, 서비스 메시 또는 API 게이트웨이 배치뿐만 아니라 마이크로서비스 자체를 안전하게 설계하고 구현하는 것이 중요하다.

3. 마이크로서비스 위협에 대한 배경지식

마이크로서비스 기반 애플리케이션 시스템의 위협에 대한 배경지식을 2장(배경지식)의 연장선에서 다룬다. 위협에 대한 배경지식을 검토하기 위해 다음의 접근방법을 적용한다.

- 각 레이어에서 일반적/잠재적인 위협을 식별할 때, 일반적인 마이크로서비스 기반 애플리케이션 배치 스택의 모든 레이어를 고려한다.
- 마이크로서비스 기반 애플리케이션 시스템에 대한 고유한 위협 요소를 식별한다.

3.1. 위협 요인의 검토 범위

일반적인 마이크로서비스 기반 애플리케이션의 배치 스택에는 6개의 레이어(하드웨어, 가상화, 클라우드, 통신, 서비스/애플리케이션, 오케스트레이션)가 존재한다.[13] 이 문서에서는 이러한 레이어들을 위협 요인으로 간주하며, 마이크로서비스 기반 애플리케이션의 위협에 대한 배경지식을 간략히 설명하기 위해 이 레이어와 관련된 몇 가지 보안 문제를 아래에서 설명한다. 다수의 위협은 다른 애플리케이션 환경에서도 공통적으로 발생한다. 즉, 마이크로서비스 기반 애플리케이션 환경에서만 발생하는 위협은 아니다.

- 하드웨어 레이어 – 멜트다운(Meltdown)/스펙터(Spectre)[8]와 같은 하드웨어의 결함이 보고되었으나, 이러한 결함은 드물다. 이 문서에서는 하드웨어를 신뢰할 수 있는 것으로 가정한다. 따라서, 하드웨어 레이어의 위협은 고려하지 않는다.

- 가상화 레이어 – 가상화 레이어에서 마이크로서비스 또는 호스팅 컨테이너에 대한 위협은 손상된 하이퍼바이저, 악의적이거나 취약한 컨테이너/VM 이미지를 사용하는 것에서 기인한다. 이러한 위협은 다른 NIST 문서에서 다루고 있기 때문에 이 문서에서는 논의하지 않는다.

- 클라우드 환경 – 가상화는 클라우드 사업자가 주로 사용하는 기술이다. 따라서, 가상화 레이어와 동일한 위협이 적용된다. 추가적으로 클라우드 사업자의 네트워크 인프라 내에는 잠재적인 위협이 존재한다. 예를 들어, 모든 마이크로서비스를 한 사업자의 클라우드 서비스에 호스팅하는 경우, 클라이언트(클라우드 외부)-마이크로서비스(클라우드 내부) 사이의 통신과는 대조적으로 프로세스 사이의 통신에 대해서는 네트워크 수준의 보안 제어가 줄어들 수 있다. 클라우드 인프라 내부의 보안 위협은 다른 NIST 문서에서 다루고 있기 때문에 이 문서에는 다루지 않는다.

- 통신 레이어 – 이 레이어는 특별하다. 마이크로서비스 기반 애플리케이션은 디자인 패러다임(느슨한 결합, API 구성)을 적용한 수많은 마이크로서비스로 구성되고, 마이크로서비스 사이에서 다른 상호작용 스타일(동기, 비동기)을 사용하기 때문이다. 마이크로서비스 핵심 기능의 다수가 이 레이어와 관련된다. 이 핵심 기능에 대한 위협은 마이크로서비스에 대한 고유한 위협으로 한정하여 3.2절에서 다룬다.

- 서비스/애플리케이션 레이어 – 이 레이어에서 위협은 악의적이거나 결함이 있는 코드에서 비롯된다. 이는 안전한 애플리케이션 개발 방법론의 영역으로 이 문서의 범위에서 벗어난다.

- 오케스트레이션 레이어 – 마이크로서비스가 컨테이너와 같은 기술로 구현되는 경우, 오케스트레이션 레이어가 필요할 수 있다. 이 레이어의 위협은 자동화 또는 설정 기능(특히 서비스를 호스팅하는 서버/컨테이너/VM의 스케줄링/클러스터링과 관련된 기능)의 비정상적인 동작과 관련된다. 따라서, 이 문서의 범위에서 벗어난다.

3.2. 마이크로서비스의 고유 위협

대부분의 최신 핵심 기능은 마이크로서비스 기반 애플리케이션의 배치 스택에서 통신 계층과 관련이 있다. 따라서, 마이크로서비스 기반 애플리케이션에 대한 전반적인 보안 전략에는 적절한 구현 옵션의 선택, 핵심 기능을 패키지로 제공하는 아키텍처 프레임워크의 식별, 마이크로서비스에 고유한 위협의 식별, 구현 옵션 중 위협 대응에 대한 범위가 포함되어야 한다.

그러나, 마이크로서비스 기반 애플리케이션은 웹 애플리케이션이 취약한 대부분의 공격[1]에 여전히 취약하다는 것을 언급해야 한다. 그리고, 이러한 공격을 방지하기 위한 다수의 통제를 마이크로서비스 코드에 계속하여 구현해야 하므로 개발자들이 API 게이트웨이 또는 서비스 메시가 마이크로서비스의 모든 보안을 제공할 것이라는 인상을 받지 않도록 해야 한다.

3.2.1. 서비스 검색 메커니즘에 대한 위협

서비스 검색 메커니즘의 기본적인 기능은 다음과 같다.

- 서비스 등록 및 등록 취소
- 서비스 검색

[1] 인젝션, 인코딩 공격, 직렬화(serialization) 공격, XSS(Cross Site Scripting), CSRF(Cross-Site Request Forgery), HTTP verb tempering[20] 등

서비스 검색 메커니즘에 대한 잠재적인 보안 위협에는 다음과 같은 것들이 포함된다.

- 시스템 내부의 악의적인 노드를 등록하고 악의적인 노드로 통신을 리다이렉트하면, 서비스 검색에 위협이 된다.
- 서비스 등록 데이터베이스가 손상되면, 서비스 요청을 잘못된 서비스로 리다이렉션시키고, 서비스 거부(DoS)를 초래한다. 또한, 악의적인 서비스로의 리다이렉션은 전체 애플리케이션 시스템에 위협을 초래한다.

3.2.2. 인터넷 기반 공격

모든 네트워크 애플리케이션 또는 분산 애플리케이션은 인터넷 기반 공격에 취약하지만, 마이크로서비스 기반 애플리케이션은 다음과 같은 이유로 인터넷 기반 공격에 더 취약하다.

- 모놀리식 애플리케이션은 더 적은 수의 RPC 인터페이스를 노출시킨다. 이에 다르게 마이크로서비스 아키텍처는 거의 항상 더 많은 수의 RPC 인터페이스를 노출시킨다. 모놀리식 애플리케이션은 다양한 비즈니스 기능을 단일 컴포넌트로 구현하는 것을 선호하고, 일반적으로 통합된 인터페이스를 노출시킨다. 마이크로서비스 아키텍처를 사용하는 애플리케이션은 작은 컴포넌트가 많고, 많은 인터페이스로 연결되거나 조정된다.
- 업스트림 컴포넌트에 대해 구현된 보안 통제가 다운스트림 컴포넌트에 직접 접근함으로써 생략된 경우, 의도하지 않게 내부 기능이 노출됨에 따라 위험이 증가한다. 시스템의 전반적인 복잡성이 증가하면, 호출자에게 어떤 조건이 적용되었지 마이크로서비스 기반 애플리케이션이 판단할 수 없기 때문에 개발자가 검사를 생략할 가능성이 증가한다.

인터넷 기반 공격에는 봇넷 공격이 포함된다. 봇넷 공격은 마이크로서비스 기반 애플리케이션에 크리덴셜 스터핑/도용, 계정 탈취, 페이지 스크래핑, 데이터 수집, 서비스 거부 등의 피해를 줄 수 있다. 물론, 이러한 피해는 다른 공격에 의해서도 발생할 수 있고, 봇넷 공격이 마이크로서비스 기반 애플리케이션만 대상으로 하는 것은 아니다.

3.2.3. 연계 고장

마이크로서비스 기반 애플리케이션은 다수의 컴포넌트가 존재하기 때문에 서비스 장애의 가능성이 높다. 컴포넌트는 배치의 관점에서 느슨하게 결합되도록 설계되어 있다. 하지만, 많은 비즈니스 트랜잭션은 다수 서비스를 차례차례로 실행하고 결과를 제공하기 때문에 논리적/기능적으로 의존적이다. 따라서, 비즈니스 트랜잭션 처리 로직에서 업스트림인 서비스에 장애가 발생하면, 장애가 발생한 서비스에 의존적인 다른 서비스 또한 대응하지 못할 것이다. 이러한 현상은 연계 고장(cascading failure)으로 알려져 있다.

4. 핵심 기능의 구현과 위협 대응에 대한 보안 전략

마이크로서비스 기반 애플리케이션 시스템의 설계 및 배치를 위한 보안 전략은 다음과 같다.

핵심 기능에 대한 구현 옵션 분석

(a) 식별 및 접근 관리(Identity and access management)
(b) 서비스 검색(Service discovery)
(c) 보안 통신 프로토콜(Secure communication protocols)
(d) 보안 모니터링(Security monitoring)
(e) 복원력/가용성 향상 기술(Resiliency or availability improvement techniques)
(f) 무결성 보증 향상 기술(Integrity assurance improvement techniques)

마이크로서비스에 고유한 위협에 대한 대응

(a) 서비스 검색 메커니즘에 대한 위협(Threats to service discovery mechanism)
(b) 인터넷 기반 공격(Internet-based attack)
(c) 연계 고장(Cascading failures)

서비스 검색은 마이크로서비스의 핵심 기능이다. 서비스 검색 메커니즘에 대한 위협은 핵심 기능에 대한 구현 옵셥 분석에서 다룬다. 유사하게, 복원력/가용성 향상에 대한 구현 옵션에서 연계 고장에 대한 대응 방법을 다룬다. 따라서, 서비스 검색 메커니즘에 대한 위협 및 연계 고장에 대해서는 별도의 보안 전략을 제시하지 않을 것이다.

4.1. 식별 및 접근 관리에 대한 전략

마이크로서비스는 API로 제공된다. 따라서, 마이크로서비스에 인증을 개시하는 방식에는 API 키(암호)를 사용하는 것이 포함된다. OAuth 2.0 프레임워크에서는 SAML 또는 OIDC(OpenID Connect)를 통해 인코딩된 인증 토큰으로 보안을 강화할 수 있다.[14] 인가와 관련하여, 프로비저닝 및 접근 정책(모든 마이크로서비스에 대한 접근을 관리)을 적용하기 위해 중앙화된 아키텍처가 필요하다. 서비스의 수가 많고, API를 사용하는 서비스가 구현되며, 실제 비즈니스 트랜잭션(예 : 고객 주문 처리 및 배송)을 지원하기 위해서 서비스의 조립이 필요하기 때문이다. 각 마이크로서비스는 다른 개발 언어 또는 플랫폼 프레임워크에서 구현될 수 있다. 따라서, 표준화된 토큰을 통해 인가 여부를 결정을 전달할 수 있는 표준화된, 플랫폼 중립적인 방법(예 : JSON 웹 토큰[1]) 또한 필요하다. 정책을 프로비저닝하고 접근을 결정하기 위해 인가 서버를 사용해야 한다.

[1] 일부 JSON 웹 토큰(JWT, JSON web token)은 JSON 포맷으로 인코딩된 OAuth 2.0 액세스 토큰이다.[15]

각 마이크로서비스의 접근 포인트에서 접근 통제 정책을 구현하는 것은 공통 정책(모든 마이크로서비스 API에 공통적으로 적용)이 동일하게 구현되었음을 보장하기 위한 추가적인 수고가 필요하다는 단점이 있다. 각 API에 구현된 보안 정책이 일치하지 않는 경우[1], 마이크로서비스 기반 애플리케이션 전체의 보안에 영향을 미칠 것이다. 또한, 각 마이크로서비스 노드에서 접근 통제를 구현하기 위한 설치 공간은 일부 노드에서 성능 문제를 초래할 수 있다. 다수 마이크로서비스 노드가 공동의 작업을 통해 트랜잭션을 수행하기 때문에, 일부 노드에서 성능에 문제가 발생하면, 여러 서비스를 거쳐 빠르게 확산될 수 있다. 이러한 요구사항을 고려하여, 마이크로서비스의 보안 식별 및 접근 관리에 대한 전략을 아래에서 설명한다.

인증 관련 보안 전략 (MS-SS-1)

- 민감한 데이터에 접근하는 마이크로서비스 API의 경우, 단순히 API 키를 사용하여 인증하는 것은 부적절하다. 중요한 API는 전자 서명(예 : Client Credential Grant) 또는 신뢰할 수 있는 출처에서 검증한 인증 토큰을 통해 접근해야 한다. 추가적으로 토큰 탈취(compromised token)로 인한 피해를 제한하기 위해 일회용 토큰(single-use token) 또는 단기 유효 토큰(short-lived token)을 사용할 수 있다.
- 인증 토큰은 핸들(handle) 기반[2]이거나, 암호학적으로 서명하거나, HMAC(Hash-based Method Authentication Code)로 보호되어야 한다.
- 애플리케이션에서 사용되는 모든 API 키는 지정된 애플리케이션과 API만 사용할 수 있도록 제한해야 한다.
- 모든 API 키의 기능은 신분 증명[3]이 제공하는 보증 수준에 비례하여 범위를 제한해야 한다.
- 마이크로서비스 관련 공유 라이브러리를 구현하여 스테이트리스(stateless) 인증 토큰(예 : JWT)을 사용하는 경우, 다음과 같은 보안 예방 조치를 준수해야 한다. (a) 토큰 만료 시간은 가능한 짧아야 한다. 토큰 만료 시간은 세션의 지속 시간을 결정하고, 활성 세션은 취소할 수 없기 때문이다. (b) 토큰 비밀 키를 라이브러리에 하드코딩하지 않아야 한다. 토큰 비밀 키는 동적으로 변경할 수 있어야 한다. 토큰 비밀 키는 데이터 볼트 솔루션에 저장해야 한다.
- OAuth 또는 OpenID Connect와 같은 표준 기술을 구현한 경우, 안전하게 배치해야 한다.[19]

1 각 API에서 일치시켜야 하는 보안 정책은 개략적인(coarse grained) 정책이다. 세부적인 정책은 마이크로서비스의 위치에서 매우 가까운 컴포넌트 또는 마이크로서비스 자체에서만 지정할 수 있다.

2 핸들 기반 토큰은 토큰 참조 정보(token reference)를 신뢰 당사자(RP, Relying Party)에게 보내어 토큰의 유효성을 검증하고, 토큰에 바인딩된 데이터를 획득한다.

3 신분 증명의 유형이 기계 주도형(machine driven)인지 인간 주도형(human driven)인지와 관련없다.

접근 관리 관련 보안 전략 (MS-SS-2)

- 모든 API와 리소스에 대한 접근 정책을 정의하고, 인가 서버에서 프로비저닝해야 한다. 개략적인 수준의 접근 정책에는 특정 IP 주소를 가진 기능을 호출할 수 있도록 허용하는 것이 명시된다. 접근 정책은 첫 번째 API 게이트웨이에서 정의되고 시행되어야 한다. 반면, 세부적인 수준의 인가(예 : 특정 마이크로서비스의 비즈니스 로직과 관련된 도메인)는 마이크로서비스의 위치에서 가장 가까운 컴포넌트(예 : 마이크로 게이트웨이) 또는 마이크로서비스 자체에서 정의되고 시행되어야 한다.
- 캐싱 메커니즘(Caching Mechanism) : 마이크로서비스에 정책 데이터를 캐시하도록 허용하는 것이 적절할 수 있다. 캐쉬된 정책 데이터는 인가 서버가 가용하지 않은 경우에만 사용해야 하며, 일정 기간이 지난 후 만료시켜야 한다.
- 인가 서버는 세부적인 정책을 지원해야 한다.
- 인가 서버의 접근 결정은 플랫폼에 중립적인 형식으로 인코딩된 표준화된 토큰(JSON 포맷으로 인코딩된 OAuth 2.0 토큰)을 통해 마이크로서비스(개별/그룹)에 전달해야 한다. 토큰은 핸들 기반 토큰(handle-based token)이나 어설션 기반 토큰(assertion-based token)을 사용할 수 있다.
- 내부 인가 토큰(마이크로 게이트웨이가 추가한 토큰 또는 각 요청에 대한 의사결정 지점에서 추가된 토큰)의 범위는 주의 깊게 통제되어야 한다. 예를 들어, 트랜잭션을 요청할 때 인증 토큰에 허용된 범위는 트랜잭션을 완료하기 위해 반드시 접근해야 하는 API 엔드포인트로 한정되어야 한다.
- API 게이트웨이에서 모든 다운스트림 마이크로서비스에 대한 인증과 접근 통제를 수행할 수 있다. 즉, 개별 서비스마다 인증과 접근 통제를 제공할 필요가 없어진다. 개별 서비스의 인증과 접근 통제를 구현하지 않은 경우, 네트워크에 특정 위치한 모든 컴포넌트는 API 게이트웨이와 API 게이트웨이의 보호를 우회하여 서비스에 익명으로 연결할 수 있다. 이는 상호 인증과 같은 통제를 통해 서비스에 직접 연결하거나 익명으로 연결되는 것으로 방지함으로써 완화할 수 있다.

4.2. 서비스 검색 메커니즘에 대한 전략

마이크로서비스는 성능 최적화 및 부하 분산을 위해 기업 또는 클라우드 인프라 곳곳에 복제하여 배치해야 할 수도 있다. 즉, 서비스는 언제든지 추가되거나 제거될 수 있으며, 네트워크의 위치에 관계없이 동적으로 배치될 수 있다. 따라서, 마이크로서비스 기반 애플리케이션 아키텍처에서 서비스 검색 메커니즘은 필수적이다. 서비스 검색 매커니즘은 일반적으로 서비스 레지스트리(registry)를 사용하여 구현된다. 온라인에서 제공되는 마이크로서비스는 서비스 등록이라는 과정에서 자신의 위치를 알리기 위해 서비스 레지스트리 서비스를 사용한다. 또한, 등록된 서비스를 검색하려는 마이크로서비스도 서비스 레지스트리 서비스를 사용한다. 따라서, 서비스 레지스트리는 기밀성/무결성/가용성을 고려하여 구성해야 한다.

SOA(Service-Oriented Architecture)에서는 중앙 집중된 ESB(Enterprise Service Bus)의 일부로 서비스 검색을 구현한다. 그러나, 마이크로서비스 아키텍처[1]에서는 경량 메시지 버스를 구현해야 한다. 경량 메시지 버스는 2.5절에서 언급한 모든 상호작용 스타일로 구현할 수 있다. 또한, 서비스 레지스트리 서비스는 다음의 두 가지 사항을 고려하여 구현한다.

(a) 클라이언트가 서비스 레지스트리 서비스에 접근하는 방법
(b) 중앙 집중 서비스 검색 vs 분산 서비스 검색

클라이언트가 서비스 레지스트리 서비스에 접근할 수 있는 방법에는 크게 클라이언트 사이드 검색 패턴과 서버[2] 사이드 검색 패턴이 있다.[9]

클라이언트 사이드 서비스 검색 패턴의 분석
클라이언트 사이드 서비스 검색 패턴은 레지스트리 인지 클라이언트(registry-aware client)를 구축하는 것으로 이루어진다. 클라이언트는 요청을 생성하기 위해 필요한 모든 서비스의 위치를 확인하기 위해 서비스 레지스트리를 조회한다. 즉, 클라이언트 사이드 서비스 검색 패턴은 클라이언트를 구현한 프로그래밍 언어/프레임워크로 검색 로직(서비스 레지스트리 조회)을 구현해야 한다.

1 마이크로서비스 아키텍처는 비즈니스 기능을 패키지로 제공하고, 컨테이너 내부에 서비스로 배치한다. 그리고, 비즈니스 기능은 API를 호출하여 서로 통신한다.

2 이 문서의 원문에서는 서비스 사이드 검색 패턴으로 명시하고 있으나, 관련문서를 포함한 대부분의 상황에서 클라이언트 사이드의 대응 표현으로 서버 사이드를 사용하기에 서버 사이드로 명시한다.

서버 사이드 서비스 검색 패턴[1]의 분석

서버 사이드 검색은 두 가지 패턴으로 구현할 수 있다. 한 패턴은 검색 로직을 고정된 위치의 전용 라우터 서비스에 위임하는 것이다. 반면, 다른 패턴은 동적 DNS 리졸버(resolver)[2] 기능을 보유한 서버(각 마이크로서비스 앞에 위치)를 이용하는 것이다. 전용 라우터 옵션에서 클라이언트는 모든 서비스를 전용 라우터 서비스에 요청한다. 전용 라우터 서비스는 서비스 레지스트리를 조회하여 클라이언트가 요청한 서비스의 위치를 찾아 해당 요청을 조회된 위치로 전달한다. 이렇게 하면, 서비스 레지스트리 서비스와 같은 인프라 서비스와 애플리케이션 서비스 간의 긴밀한 결합이 제거된다. DNS 리졸버 패턴에서 각 마이크로서비스는 서비스 레지스트리를 조회하기 위해 내장된 DNS 리졸버를 사용하여 자체적으로 서비스 검색을 완료한다. DNS 리졸버는 가용한 서비스 인스턴스와 인스턴스의 위치(IP 주소)의 테이블을 관리한다. 이 테이블을 최신 상태로 유지하기 위해, 비동기/비차단 DNS 리졸버는 서비스를 검색하기 위한 DNS 서비스 레코드[3]를 사용하여 서비스 레지스트리를 정기적(몇 초 단위)으로 조회한다. DNS 리졸버를 통한 서비스 검색 기능은 백그라운드 작업으로 실행되다. 따라서, 서비스 인스턴스가 요청을 생성할 때, 모든 상대 마이크로서비스의 위치(URL)를 즉시 사용할 수 있다.[2]

좋은 전략은 서버 사이드 서비스 검색 패턴과 클라이언트 사이드 서비스 검색 패턴을 혼합하여 사용하는 것이다.[9] 서버 사이드 서비스 검색 패턴은 퍼블릭 API에 접근하는데 사용하고, 클라이언트 사이드 서비스 검색 패턴은 클러스터 내부의 상호작용을 처리한다.

중앙 집중 서비스 레지스트리 vs 분산 서비스 레지스트리

중앙 집중 서비스 레지스트리의 경우, 단일 지점에서 서비스를 등록하고, 단일 레지스트리를 사용하여 서비스를 검색한다. 중앙 집중 서비스 레지스트리의 단점은 SPOF(single point of failure, 단일 장애점)이다.[13] 하지만, 데이터의 일관성에는 문제가 없다. 분산 서비스 레지스트리의 경우, 다수의 서비스 레지스트리 인스턴스가 존재할 수 있고, 서비스는 어떤 인스턴스에라도 등록할 수 있다. 분산 서비스 레지스트리의 단점은 짧은 시간 동안 여러 서비스 레지스트리 사이에 데이터 불일치가 발생할 수 있다는 것이다. 사실상, 다수 서비스 레지스트리 인스턴스 사이의 일관성 문제는 한 인스턴스에서 다른 인스턴스로 전파하거나, 프로세스에 첨부되는 데이터(피기배킹, piggybacking)를 통해 해결할 수 있다.

서비스 검색에 사용되는 패턴에 관계없이 서비스 검색 기능을 안전하게 배치하기 위해서는 다음의 서비스 레지스트리 구성 요구 사항을 충족해야 한다.

---

1 이 문서의 원문에서는 "서버 사이드 서비스 검색 패턴" 대신 "서비스 사이드 서비스 검색 패턴"이라는 용어를 사용하고 있으나, 통상 "클라이언트"의 반대 개념으로 "서버"를 사용하고, 다른 관련문서에서도 "서버 사이드 서비스 검색 패턴"으로 사용하고 있어 "서버 사이드 서비스 검색 패턴"으로 번역하였다.

2 동적 DNS 리졸버는 DNSSEC(Domain Name System Security Extensions)의 권한 네임 서버(authoritative server)와 함께 동작한다.

3 예 : 서비스 리소스 레코드(SRV RRs, Service Resource Records)

서비스 레지스트리 구성 관련 보안 전략 (MS-SS-3)

- 전용 서버 또는 서비스 메시 아키텍처의 일부로 서비스 레지스트리 기능을 제공해야 한다.
- 서비스 레지스트리 서비스는 가용성과 복원력을 보장하기 위해, QoS(Quality of Service) 설정할 수 있도록 구성된 네트워크에 위치해야 한다.
- 애플리케이션 서비스와 서비스 레지스트리 간의 통신은 HTTPS나 TLS(Transport Layer Security)와 같은 안전한 통신 프로토콜을 사용해야 한다.
- 서비스 레지스트리는 정당한 서비스만 등록, 갱신, 서비스를 검색하기 위한 데이터베이스 조회를 수행한다는 것을 보장하기 위해 유효성을 검사해야 한다.
- 서비스 등록/등록 취소 기능에 대해 마이크로서비스가 독립적으로 서비스되어도 문제가 발생하지 않는 범위(bounded context)와 느슨한 결합 원칙을 준수해야 한다. 즉, 애플리케이션 서비스는 서비스 레지스트리 서비스 등 인프라 서비스와 긴밀하게 결합되는 것을 지양해야 한다. 그리고, 서비스 자체 등록/등록 취소 패턴[1]을 지양해야 한다. 애플리케이션 서비스가 중단되거나 실행 중이지만 요청을 처리할 수 없는 경우, 등록 취소는 전체 프로세스의 무결성에 영향을 미친다. 따라서, 서드 파티 등록 패턴을 사용하여 애플리케이션 서비스의 등록/등록 취소를 수행해야 한다. 그리고, 애플리케이션 서비스는 서비스 레지스트리에 대한 조회만 수행해야 하도록 제한해야 한다.
- 서드파티 등록 패턴을 구현한다면, 등록/등록 취소는 애플리케이션 서비스에 대한 헬스 체크가 수행된 후에만 이루어져야 한다.
- 대규모 마이크로서비스 애플리케이션의 경우, 분산 서비스 레지스트리를 배치해야 한다. 다수 서비스 레지스트리 인스턴스 간의 데이터 일관성을 유지하기 위해 주의를 기울여야 한다.

1 이 문서의 원문에서는 설명하고 있지 않지만, 서비스 등록/등록 취소 패턴은 "자체 등록/등록 취소 패턴"과 "서드파티 등록/등록 취소 패턴"으로 구분할 수 있다. "자체 등록/등록 취소 패턴"은 각 서비스가 서비스 레지스트리에 등록(시작 시)하고 등록 취소(종료 시)한다. 반면, "서드파티 등록/등록 취소 패턴"은 서비스 매니저(Service Manager)가 서비스 레지스트리에 각 서비스를 등록하고 등록 취소한다. 서비스의 복원력 관점에서 서드파티 등록/등록 취소 패턴이 더 우수하다. 자체 등록/등록 취소 패턴에서는 서비스의 장애를 서비스 레지스트리가 인지하지 못 할 수 있다.

4.3. 안전한 통신 프로토콜에 대한 전략

클라이언트-서비스, 서비스-서비스 사이의 안전한 통신은 마이크로서비스 기반 애플리케이션의 작업에 있어 중요하다.

그러나, 보안 서비스와 관련된 특정 전략(인증, 보안 연결 설정 등)을 개별 마이크로서비스 노드에서 처리할 수 있다. 예를 들어, 패브릭 모델(fabric model)에서 각 마이크로서비스 인스턴스는 SSL 클라이언트 및 SSL 서버 기능을 가진다. 즉, 각 마이크로서비스는 SSL/TLS 엔드포인트이다. 따라서, 전반적인 애플리케이션 관점에서 서비스 또는 프로세스 간 통신에 SSL/TLS 연결을 사용할 수 있다. 각 서비스 간 요청 전에 SSL/TLS 연결을 동적으로 생성하거나, 킵얼라이브(keep-alive) 연결로 생성할 수 있다. 킵얼라이브 연결 구조에서 "서비스 A"의 인스턴스가 "서비스 B"의 인스턴스로 처음 요청할 때 SSL/TLS 핸드셰이크(handshake)가 수행되며, SSL/TLS 핸드셰이크가 완료한 후 "서비스 A"는 연결을 생성한다. 그러나, 서비스 인스턴스는 "서비스 B"가 "서비스 A"의 요청에 응답한 후에도 연결을 종료하지 않는다. 오히려 동일한 연결이 향후 요청에서 재사용된다. 킵얼라이브 연결 구조의 장점은 각 요청 동안 초기 SSL/TLS 핸드셰이크 수행에 수반되는 고비용의 오버헤드(overhead)를 피할 수 있다는 것과 기존 연결을 수천 건의 후속 서비스 간 요청에 재사용할 수 있다는 것이다. 따라서, 모든 요청에 대해 지속적이고 안전하게 서비스 간 네트워크 연결이 가능하다.

안전한 통신에 대한 보안 전략 (MS-SS-4)

- 클라이언트가 대상 서비스를 직접 호출하지 않도록 구성해야 하며, 단일 게이트웨이 URL을 가리키도록 구성해야 한다.
- 서버-서버 통신뿐만 아니라 클라이언트-API 게이트웨이 통신도 상호 인증 후 수행되어야 하며, 암호화(예 : mTLS 프로토콜)해야 한다.
- 상호 작용이 빈번한 서비스는 킵얼라이브 TLS 연결을 생성해야 한다.

4.4. 보안 모니터링에 대한 전략

한 개의 서버(또는 부하 분산을 위한 일부 복사본)에서 실행되는 모놀리식 애플리케이션을 모니터링하는 것과 비교했을 때, 마이크로서비스 기반 시스템은 수 많은 서비스[1]를 모니터링해야 한다. 또한, 시스템에서 중요한 트랜잭션은 최소 2개 이상의 서비스에 영향을 미친다.

보안 모니터링에 대한 보안 전략 (MS-SS-5)

- 게이트웨이와 서비스 모두에서 부적절한 행위(예 : 소유 토큰[2] 재사용 공격, 인젝션 공격)를 탐지/경고/대응하기 위해 보안 모니터링을 수행해야 한다. 또한, 입력 검증 오류, 추가 파라미터 오류, 충돌, 코어 덤프를 기록해야 한다. 이러한 기능을 제공하는 소프트웨어로는 OWASP(the Open Web Application Security Project) 앱센서(AppSensor)가 있다. 앱센서는 게이트웨이, 서비스 메시, 마이크로서비스에 구현될 수 있다.
- 중앙 대시보드는 다양한 서비스와 서비스를 연결하는 네트워크 세그먼트의 상태를 표시한다. 최소한 대시보드는 입력 검증 실패, 인젝션 공격 시도가 명확한 파라미터와 같은 보안 파라미터를 보여주어야 한다.
- 비즈니스 로직의 의사결정 결과, 접근 시도 등의 관점에서 정상적이고 위험하지 않은 행위의 기준을 만들어야 한다. IDS(Intrusion Detection System)는 이 기준을 위반하는 행위를 탐지할 수 있도록 충분한 기능을 보유해야 하며, 적절한 위치에 배치해야 한다.

1 다른 서버에서 실행되는 각 서비스는 이기종의 애플리케이션 플랫폼에 호스팅될 수 있다.

2 소유 토큰(bearer token)은 토큰을 소유한 모든 클라이언트가 사용할 수 있는 유형의 토큰을 말한다. 토큰을 소유하고 있다면 토큰을 사용하는데 제한이 없다. 일부에서는 전달 토큰, 무기명 토큰 등으로 번역하기도 한다. 이와 대비되는 유형은 입증 토큰(proof token)으로 특정 클라이언트에서만 사용할 수 있는 토큰이다. 토큰을 사용할 때마다 클라이언트는 토큰의 인증된 사용자임을 입증해야 한다.

4.5. 가용성/복원력 향상 전략

마이크로서비스 기반 애플리케이션의 전반적인 보안을 향상시키기 위해 특정 중요 서비스의 가용성 또는 복원력을 향상시켜야 한다. 일반적으로 배치되는 기술은 다음과 같다.

- 서킷브레이커 기능
- 부하 분산
- 사용량 제한(스로틀링)

4.5.1. 서킷브레이커 구현 옵션 분석

연계 고장을 방지하거나 최소화하기 위해 서킷브레이커를 사용하는 것이 일반적인 전략이다. 서킷브레이커는 지정된 임계값을 초과하여 오류가 발생하는 컴포넌트(마이크로서비스)로 데이터를 전달하지 않는다. 이러한 방식은 fail-fast 원칙[1]으로 알려져 있기도 하다. 장애가 발생한 서비스를 빠르게 오프라인으로 전환하기 때문에, 연계 고장의 발생을 최소화한다. 동시에 장애가 발생한 컴포넌트의 로그를 분석하고, 오류를 수정하고, 마이크로서비스를 업데이트한다. 서킷브레이커는 클라이언트, 서비스 또는 프락시에 배치할 수 있다.[9]

클라이언트 사이드 서킷브레이커 옵션 : 이 옵션에서 각 클라이언트는 개별 서킷브레이커 또는 클라이언트가 호출하는 개별 외부 서비스를 갖는다. 클라이언트의 서킷브레이커가 서비스(이런 서비스를 "오픈 스테이트"라고 부른다.) 호출을 차단하기로 결정했다면, 어떤한 메시지도 서비스로 보내지 않으며, 이후 네트워크에서 통신 트래픽은 감소한다. 또한, 서킷브레이커 기능은 마이크로서비스에 구현할 필요가 없다. 이는 고가의 자원이 필요하지 않기 때문에 서비스를 효율적으로 구현할 수 있게 한다. 그러나, 클라이언트에 서킷브레이커를 배치하는 것은 보안의 관점에서 두 가지 단점이 존재한다. 첫 번째, 서킷브레이커 코드가 적절하게 실행되도록 클라이언트에 많은 신뢰를 부여해야 한다. 두 번째, 서킷브레이커 운영의 전체적인 무결성을 위험하게 한다. 클라이언트 한 개의 정보만으로 서비스의 가동 상태를 판단하기 때문이다. 모든 클라이언트가 수신한 서비스의 응답을 종합하여 서비스의 상태를 결정하지 않고, 클라이언트 한 개에서 서비스를 호출한 빈도에 따라 상태를 결정한다.

서버 사이드 서킷브레이커 옵션 : 이 옵션에서, 마이크로서비스의 내부 서킷브레이커는 모든 클라이언트의 호출을 처리하고 서비스 호출 여부를 결정한다. 이 옵션의 보안 장점은 서킷브레이커의 기능을 구현하기 위해 클라이언트를 신뢰할 필요가 없다는 것이다. 서킷브레이커 서비스는 모든 클라이언트의 모든 호출 빈도를 전체적으로 파악할 수 있기 때문에 편리하게 처리할 수 있는 수준(예: 일시적인 부하 경감)으로 요청을 조절할 수 있다.

1 장애나 오류 나타낼 가능성이 있는 모든 상태를 인터페이스에서 즉시 보고하는 시스템 설계 방식이다.

프락시 서킷브레이커 옵션 : 이 옵션에서 서킷브레이커는 클라이언트와 마이크로서비스 사이에 위치한 프락시 서비스에 배치되며, 모든 송수신 메시지를 처리한다. 각 마이크로서비스 별로 프락시를 두거나, 다수 서비스에 대해 단일 프락시(보통 API 게이트웨이에 구현)를 배치할 수 있다. 또한, 프락시에는 클라이언트 사이드 서킷브레이커와 서버 사이드 서킷브레이커를 모두 포함한다. 이 옵션의 보안 장점은 클라이언트의 코드 뿐만 아니라 서비스의 코드도 수정할 필요가 없다는 것이다. 이는 서킷브레이커 기능 뿐만 아니라 이와 유사한 코드와 관련된 신뢰 및 무결성 보증 문제를 피할 수 있다. 또한, 이 옵션은 결함이 있는 서비스에 대한 클라이언트의 복원력을 높이고, 서비스 거부(DoS, Denial of Service)를 초래할 수 있는 클라이언트의 과도한 요청[9]으로부터 서비스를 보호하는 등의 추가적인 보호를 제공한다.

서킷브레이커 구현에 대한 보안 전략 (MS-SS-6)

- 신뢰할 수 있는 컴포넌트를 프락시만으로 제한하기 위해 프락시 서킷브레이커 옵션을 적용해야 한다. 프락시 서킷브레이커 옵션은 클라이언트와 마이크로서비스에 신뢰(예 : 임계값을 설정하고 설정된 임계값을 기준으로 요청을 차단)를 부여할 필요가 없다. 클라이언트와 마이크로서비스는 다수가 존재하기 때문이다.

4.5.2. 부하 분산에 대한 전략

부하 분산은 마이크로서비스 기반 애플리케이션에서 필수적인 기능 모듈이며, 주요 목적은 서비스의 부하를 분산하는 것이다. 서비스 이름은 네임스페이스와 관련된다. 네임스페이스는 다수 인스턴스가 동일한 서비스를 제공하는 것을 지원한다. 즉, 동일한 서비스의 다수 인스턴스가 동일한 네임스페이스를 사용한다.[13] 서비스 부하 분산을 위해 로드밸런서는 라운드로빈(요청을 서비스 인스턴스에 할당하기 위한 순환 패턴)과 같은 알고리즘을 사용하여 요청된 네임스페이스에서 하나의 서비스 인스턴스를 선택한다.

부하 분산에 대한 보안 전략 (MS-SS-7)

- 부하 분산 기능을 지원하는 모든 프로그램을 개별적인 서비스 요청으로부터 분리해야 한다. 예를 들어, 서비스에서 부하 분산 풀(pool)을 결정하기 위해 헬스 체크를 수행하는 프로그램은 백그라운드에서 비동기로 실행해야 한다.
- 로드 밸런서와 마이크로서비스 플랫폼 사이의 네트워크 연결을 보호하기 위해 주의를 기울여야 한다.
- DNS 리졸버가 마이크로서비스 앞에 배치되어 가용한 마이크로서비스 인스턴스의 테이블을 제공하는 경우(DNS 리졸버를 사용한 서버 사이드 서비스 검색 패턴), DNS 리졸버는 단일 목록을 호출한 마이크로서비스에게 표시하기 위해 헬스체크 프로그램과 동시에 작동해야 한다.

4.5.3. 사용량 제한(스로틀링)

사용량 제한의 기본적인 목적은 서비스가 과도하게 구독되어 가용성에 영향을 미치지 않도록 보장하는 것이다. 한 클라이언트가 요청의 비율이 증가할 때, 서비스는 다른 클라이언트에 대한 대응을 계속한다. 이는 클라이언트에 대해 정해진 시간 내 서비스 호출 빈도를 제한하도록 설정함으로써 해결한다. 제한을 초과했을 때, 클라이언트는 애플리케이션과 관련된 응답을 받지 못한다. 대신, 허가된 비율을 초과했다는 알림, 초과한 요청의 수, 클라이언트가 응답 수신을 재개하기 위해 카운터가 초기화될 시간을 수신한다. 사용량 제한의 부수적인 목적은 DoS 공격의 영향을 완화하는 것이다. 사용량 제한의 개념과 밀접하게 관련이 있는 것은 할당량 관리 또는 조건부 사용량 제한이다. 제한은 인프라의 제한/요구사항보다 애플리케이션의 요구사항에 기반하여 결정된다.

사용량 제한에 대한 보안 전략 (MS-SS-8)

- 애플리케이션 사용에 대한 할당량 또는 제한은 인프라 및 애플리케이션 요구사항 모두에 기반해야 한다.
- 명확한 API 사용 계획에 기반하여 제한을 결정해야 한다.
- 높은 보안이 요구되는 마이크로서비스는 재사용 공격 탐지(replay detection) 기능을 구현해야 한다. 위험을 기반으로 항상 재사용 공격을 탐지하거나, 임의로 탐지하도록 설정할 수 있다.

4.6. 무결성 보증 전략

마이크로서비스 기반 애플리케이션의 맥락에서 무결성 보증 요구사항은 두 가지 맥락에서 발생한다.

- 마이크로서비스의 새로운 버전을 시스템에 반영
- 트랜잭션 중 세션 지속(session persistence)

새로운 릴리즈 반영 모니터링 : 새로운 버전의 마이크로서비스가 릴리즈되면, 점진적인 프로세스에 따라 반영되어야 한다. (a) 모든 클라이언트가 새로운 버전을 사용할 준비가 되어 있지 않을 수 있고, (b) 광범위한 테스트에도 불구하고 모든 시나리오/유스케이스에 대한 새로운 버전의 행위가 비즈니스 프로세스의 예상에 부합하지 않을 수 있기 때문이다. 이 상황을 해결하기 위해 카나리 릴리즈(canary release)를 주로 채택한다.[4] 카나리 릴리즈에서는 제한된 수의 요청만이 새로운 버전으로 전달되며, 나머지 요청은 기존 운영중인 버전으로 전달된다. 관찰 기간이 지난 후 새 버전이 성능과 무결성 기준을 만족한다는 것을 보증하며, 모든 요청이 새로운 버전으로 전달된다.

마이크로서비스의 새로운 버전 반영에 대한 무결성 보증 전략 (MS-SS-9)

- 서비스의 기존 버전과 새로운 버전에 대한 트래픽은 통제된 방법으로 새로운 버전으로 전환되는지 모니터링하고, 카나리 릴리즈와 관련한 위험을 모니터링 하기 위해 API 게이트웨이와 같은 중앙 노드를 통해 전달되어야 한다. 기존 버전과 최신 버전을 호스팅하는 노드 모두에 대한 보안을 모니터링해야 한다.
- 기존 버전에서 새로운 버전으로 트래픽을 지속 증가시켜야 한다.
- 새로운 버전의 성능과 기능 정확도는 새로운 버전으로 트래픽을 증가시키는 요소여야 한다.
- 카나리 릴리즈를 설계할 때, 기존 버전 또는 새로운 버전에 대한 클라이언트의 선호도를 고려해야 한다.

세션 지속 : 클라이언트는 특정 서비스로 다수의 요청을 보내어 트랜잭션을 완성하기 때문에, 클라이언트 세션에서 모든 요청을 동일한 업스트림 마이크로서비스 인스턴스로 보내는 것이 중요하다. 해당 세션에서 모든 요청은 업스트림 서비스 인스턴스가 동일해야 한다. 이 요구사항을 세션 유지라고 부른다. 이 요구사항을 훼손할 수 있는 잠재적인 상황은 마이크로서비스가 자신의 상태를 로컬에 저장하고, 개별 요청을 처리하는 로드밸런서가 현재 사용자 세션을 다른 마이크로서비스나 서비스로 전송하는 경우이다. 세션 유지를 구현하기 위한 방법 중 하나는 스티키 쿠키(sticky cookie)이다. 이 방법에는 업스트림 마이크로서비스 그룹에서 특정 클라이언트로 송신하는 첫 번째 응답에 세션 쿠키를 추가하여, 응답을 생성한 서버를 식별하는 메커니즘이 있다. 클라이언트의 후속 요청에는 이 세션 쿠키가 포함되며, 동일한 메커니즘을 사용하여 요청을 동일한 업스트림 서버로 전달한다.[16]

세션 지속에 대한 무결성 보증 전략 (MS-SS-10)

- 클라이언트에 대한 세션 정보를 안전하게 저장해야 한다.
- 바인딩된 서버 정보를 전달하는데 사용되는 아티팩트(세션 쿠키)를 보호해야 한다.
- 내부 인가 토큰을 사용자에게 제공하지 않아야 한다. 또한, 사용자의 세션 토큰을 정책 결정에 사용하기 위한 목적으로 게이트웨이를 넘겨 전달하지 않아야 한다.

4.7. 인터넷 기반 공격 대응

봇넷 등 모든 유형의 인터넷 기반 공격으로부터 보호하는 것은 불가능하지만, 마이크로서비스 API에는 악의적인 봇넷을 탐지할 수 있는 기능과 크리덴셜 스터핑 공격과 크리덴셜 도용 공격에 대한 예방/탐지 기능이 제공되어야 한다. 각 마이크로서비스를 독립적으로 호출할 수 있고, 자체적으로 인증 정보를 저장하고 있는 애플리케이션이라면 이러한 기능의 제공은 특히 중요하다. 크리덴셜 도용 공격은 오프라인 위협 분석 또는 런타임 솔루션을 사용하여 탐지할 수 있다.[17] 봇넷 공격은 봇 매니저 또는 웹 방화벽(WAF, Web Application Firewall)의 추가 기능을 통해 탐지할 수 있다.

크리덴셜 도용/크리덴셜 스터핑 공격 예방을 위한 보안 전략 (MS-SS-11)

- 크리덴셜 도용에 대한 예방 전략은 오프라인 전략보다 런타임 전략이 효과적이다. 특정 위치(예 : IP 주소)에서 지정된 시간 간격 동안 로그인 시도 횟수에 대한 임계값을 설정해야 한다. 임계값을 초과하면, 인증/인가 서버는 예방을 위한 조치를 실행해야 한다. 소유 토큰(bearer token)을 사용하는 경우, 토큰 재사용의 예방 및 탐지를 위해 이 기능이 반드시 존재해야 한다.
- 크리덴셜 스터핑 탐지 솔루션은 탈취된 크리덴셜 데이터베이스에 대비하여 사용자 로그인을 검사하고, 정당한 사용자에게 크리덴셜이 탈취되었음을 경고하는 기능을 보유해야 한다.
- DoS 공격을 탐지하고, 서비스가 접근이 불가해지기 전에 경고하도록 IDS와 네트워크의 경계에 위치한 장비(boundary device)를 구성해야 한다.
- 악성코드로 인한 위협을 완화하기 위해 파일 업로드, 컨테이너의 메모리 및 파일 시스템을 검사하도록 서비스 호스트를 구성해야 한다.

5. 마이크로서비스 아키텍처 프레임워크에 대한 보안 전략

이 문서에서는 마이크로서비스 기반 애플리케이션 시스템의 두 가지 주요 아키텍처 프레임워크인 API 게이트웨이와 서비스 메시를 고려한다. API 게이트웨이를 구현할 때 주요한 고려사항은 API 게이트웨이를 호스팅하기에 적절한 플랫폼을 선택하고, 엔터프라이즈 인증/인가 프레임워크와의 적절하게 통합 및 구성하며, 보안 모니터링 및 분석을 위해 API 게이트웨이를 통과하는 트래픽을 안전하게 사용하는 것이다.

API 게이트웨이 구현에 대한 보안 전략 (MS-SS-12)

- API 게이트웨이와 계정 관리 애플리케이션을 통합하여 크리덴셜을 제공한 후, API를 활성화해야 한다.
- API 게이트웨이를 통해 계정 관리 애플리케이션을 호출할 경우, ID 제공자(IdP, identity provider)와 통합하기 위한 커넥터가 제공되어야 한다.
- API 게이트웨이는 클라이언트의 요청에 대해 액세스 토큰을 생성할 수 있는 아티팩트(예 : OAuth 2.0 인가 서버)에 대한 커넥터를 보유해야 한다.
- 공격(예 : DoS, 악의적인 행위)을 탐지하고 성능 저하의 원인을 밝혀내기 위해 모든 트래픽 정보를 모니터링 및 분석 애플리케이션으로 안전하게 전송해야 한다.
- 분산 게이트웨이를 배치하는 경우, 게이트웨이 간 토큰을 변환(교환)하는 서비스가 존재해야 한다.[18] 첫 번째 게이트웨이에서 제공하는 토큰은 광범위한 사용 권한을 가져야 한다. 반면, 내부 게이트웨이(또는 마이크로 게이트웨이)에서 제공하는 토큰은 범위가 좁고, 특정한 권한을 가져야 한다. 또는 대상 마이크로서비스 플랫폼에 적합한 완전히 다른 형태의 토큰이어야 한다. 이는 최소 권한의 법칙을 구현하는데 도움을 준다.

서비스 메시를 구현하면 컨트롤 플레인의 설정 파라미터가 정확하게 설정되어 보안 정책의 취지를 충족하고, 서비스 메시에서 취약점이 발생하지 않는다는 것을 보장해 준다.

서비스 메시 구현에 대한 보안 전략 (MS-SS-13)

- 서비스 간 통신에 사용할 프로토콜을 지정하고, 애플리케이션의 요구사항에 따라 서비스 상이의 트래픽 부하를 지정하기 위한 정책을 지원해야 한다.
- 모든 서비스에 대해 접근 통제 정책을 항상 실행하도록 기본 설정되어야 한다.
- 권한 상승을 초래할 수 있는 설정(예 : 서비스 역할 권한, 서비스 사용자 계정에 대한 서비스 역할 바인딩)은 지양해야 한다.
- 서비스 메시는 컴포넌트의 리소스 사용 한계를 지정할 수 있는 기능이 있어야 `한다. 이 기능이 없으면, 컴포넌트가 전체 마이크로서비스 애플리케이션의 복원력과 가용성에 영향을 미칠 수 있다.

- 서비스 메시는 환경 분석 결과(요청된 분석 결과 포함)를 수집하고, 모니터링을 위해 중앙화된 서비스로 전송하도록 설정할 수 있는 기능이 있어야 한다. 정책은 멀티 클러스터 마이크로서비스 환경에 대한 고가용성과 복원력을 보장하기 위해 싱글 서비스 메시 또는 멀티 서비스 메시(각 서비스 메시는 자체 컨트롤 플레인을 가짐)를 지정하는 것을 감안해야 한다.
- 중요도가 높은 마이크로서비스 기반 애플리케이션이라면, 오케스트레이션 플랫폼 내부에서 서브넷팅(subnetting)으로 네트워크를 분할(segmentation)해야 한다. 서비스 메시 레이어는 전체적으로 세션 제한으로 네트워크를 분할한다. 서브넷팅은 사이드카 프록시(네트워크 트래픽을 보호/차단하기 위해 사용)를 우회하는 악의적인 행위자에 의한 위협에 대응하기 위한 보완책이다.

부록 A. 모놀로식-마이크로서비스 기반 애플리케이션의 차이점

A.1. 설계와 배치의 차이점

개념적으로 모놀로식 애플리케이션 아키텍처는 한 개의 거대한 산출물을 생성하고, 전체적으로 배치되어야 한다. 반면 마이크로서비스 기반 애플리케이션은 서비스 또는 마이크로서비스라 불리는 독립적이고 느슨하게 결합된 다수의 실행 파일이 있다. 개별 서비스는 독립적으로 배치된다. 모놀로식 애플리케이션의 경우, 전체 애플리케이션의 특정 기능을 변경하면 다시 컴파일해야 한다. 경우에 따라 배치하기 전에 전체 애플리케이션을 다시 테스트해야 한다. 그러나, 마이크로서비스는 단지 관련된 서비스를 수정하고 재배치한다. 한 서비스의 변경이 다른 서비스의 기능에 논리적으로 영향을 미치지 않는다는 것을 보장하는 독립적인 특성 때문이다. 모놀리식 애플리케이션의 경우, 부하가 증가(사용자의 수 또는 애플리케이션 사용 빈도의 증가)하면 전체 애플리케이션에 리소스를 할당해야 한다. 반면, 마이크로서비스는 성능이 기대치보다 적은 서비스에 선택적으로 자원을 증가시킬 수 있다. 따라서, 확장에 있어 유연성을 갖는다.

일부 모놀로식 애플리케이션을 모듈식으로 구성할 수 있다. 그러나, 의미적 또는 논리적 모듈성은 갖지 않을 수 있다. 모듈 구조란, 다른 벤더에서 제공되었을 수 있는 많은 수의 컴포넌트와 라이브러리로 구축하는 방법 및 일부 컴포넌트(예 : 데이터베이스)가 네트워크에 배포되는 방법을 말한다.[13] 모놀리식 애플리케이션에서 API의 설계와 사양은 마이크로서비스 아키텍처와 유사하다. 그러나, 모듈식으로 설계(고전적인 모듈식 설계)된 모놀로식 애플리케이션와 마이크로서비스 기반 애플리케이션의 차이는 마이크로서비스 기반 애플리케이션에서는 개별 API가 네트워크에 노출되며, 따라서 독립적으로 호출할 수 있고 재사용할 수 있다는 것이다.

모놀리식 애플리케이션과 마이크로서비스 기반 애플리케이션의 차이점은 다음과(표 3)과 같다.

모놀로식 애플리케이션	마이크로서비스 기반 애플리케이션
전체로 배치되어야 한다.	서비스를 독립적/선택적으로 배치할 수 있다.
애플리케이션의 사소한 변경에도 전체 애플리케이션을 재배치해야 한다.	수정된 서비스만 재배치가 필요하다.
확장이 필요한 경우, 애플리케이션 전체에 리소스를 할당해야 한다.	개별 서비스에 대해 더 많은 리소스를 할당하여 선택적으로 확장할 수 있다.
API 호출은 로컬에서 발생한다.	네트워크에 공개된 API는 독립적인 호출과 재사용이 가능하다.

표 3 : 모놀리식-마이크로서비스 기반 애플리케이션의 논리적 차이점

A.1.1. 설계 및 배치의 차이점을 설명하기 위한 애플리케이션 예제

다음의 소규모 온라인 쇼핑(retail) 애플리케이션 예제로 위에서 언급한 설계와 배치의 차이를 설명한다. 이 애플리케이션의 주요 기능은 다음과 같다.

- 판매자가 제공하는 제품의 카탈로그를 사진, 제품 번호, 이름, 가격과 함께 표시한다.
- 해당 세션에서 주문한 모든 아이템을 저장하는 공간을 생성하고, 고객 정보(예 : 이름, 주소)와 상세 주문 정보(예 : 제품 이름, 수량, 가격)를 수집하여 고객의 주문을 처리한다.
- 전체 배송 확인 서류(예 : 배송할 전체 패키지, 각 아이템의 수량, 선적 우선순위, 선적 주소)를 명시하고, 배송을 위한 주문서를 작성한다.
- 고객에게 청구서를 보내고, 신용카드 또는 은행 계좌로 결제할 수 있다.

이 온라인 쇼핑 애플리케이션의 설계에서 모놀로식-마이크로서비스 기반 애플리케이션의 차이점은 다음(표 4)과 같으며, 각각의 애플리케이션 구성도는 그림 1 및 그림 2와 같다.

애플리케이션 구조	모놀로식	마이크로서비스 기반
기능 모듈 간 통신	모든 통신은 프로시저 호출이나 내부 자료 구조(예 : 소켓)의 형태로 이루어진다. 주문 처리 모듈은 배송 처리 모듈로 프로시저를 호출하고, 성공적으로 완료될 때까지 대기(차단형 동기 통신)한다.	배송 기능과 주문 처리 기능을 각 독립된 서비스로 설계한다. 통신은 네트워크를 통해 API 호출로 이루어지며, 웹 프로토콜을 사용한다. 주문 처리 마이크로서비스는 (a) 배송 마이크로서비스를 요청하고, 응답을 기다린다. 또는 (b) 발송할 주문의 세부내용을 메시지 큐에 넣어 이벤트를 구독한 배송 마이크로서비스가 비동기식으로 선택하게 한다.
변경 및 개선사항 처리 (예 : 직불카드를 추가하기 위해 청구서 발송 모듈의 수정 필요)	필요한 변경을 수행한 후, 전체 애플리케이션을 다시 컴파일하고 다시 배포해야 한다.	청구서 발송 기능은 별도의 마이크로서비스로 설계된다. 따라서, 그 서비스만 다시 컴파일하고 배포할 수 있다.
애플리케이션 확장, 증가된 자원 할당 (예 : 더 높은 부하를 처리하기 위해 주문 처리 모듈에 더 많은 자원 할당이 필요)	주문 처리 기능은 배송 또는 청구서 발송 기능에 비해 트랜잭션에 더 오랜 시간이 소요된다. 더 많은 메모리와 CPU를 갖는 서버(수직적 확장)를 전체 애플리케이션에 대해 배치해야 한다.	주문 처리 마이크로서비스가 배치된 하드웨어에 자원을 증가시키는 것으로 충분하다. 또한, 주문 처리 마이크로서비스의 인스턴스 수를 늘려 부하를 분산할 수 있다.

표 4 : 모놀로식-마이크로서비스 기반 애플리케이션 간 애플리케이션 구조의 차이점

애플리케이션 구조	모놀로식	마이크로서비스 기반
개발과 배치 전략	개발팀이 개발하고, QA팀이 필요한 테스트 후 배치 작업을 인프라팀으로 전달한다. 인프라팀은 배치와 관련하여, 적합한 자원을 할당한다.	단일 DevOps 팀이 각 마이크로서비스에 대한 전체 생명주기(개발→배치)를 담당한다. 각 마이크로서비스는 단일한 기능을 갖고, 기능에 최적화된 플랫폼(예 : OS, 라이브러리)이 내장된 상대적으로 작은 모듈이다.

표 4 : 모놀로식-마이크로서비스 기반 애플리케이션 간 애플리케이션 구조의 차이점

A.2. 런타임 차이점

모놀리식 애플리케이션은 단일 컴퓨터 노드로 실행되며, 노드는 전체 시스템 또는 어플리케이션의 상태를 인지한다. 마이크로서비스 환경의 경우, 서비스를 제공하는 노드의 집합으로 애플리케이션을 설계한다. 노드는 다른 노드와 협조할 필요없이 운영되기 때문에 개별 노드는 전체 시스템 상태를 알 수 없다. 전역 정보 또는 전역 변수의 값이 없는 경우, 개별 노드는 가용한 로컬 정보에 기초하여 결정을 내린다. 노드의 독립성은 한 노드의 고장이 다른 노드에 영향을 미치지 않는다는 것을 의미한다. 모놀리식 애플리케이션은 데이터베이스 또는 데이터 저장소를 공유한다. 이와 다르게 마이크로서비스 아키텍처는 각 서비스가 자체 데이터 저장소를 갖는 패턴으로 배치할 수 있다. 여러 상황에서 서비스 간의 상호작용이 적절하게 설계되지 않으면, 데이터베이스의 무결성에 영향을 줄 수 있는 분산된 트랜잭션이 필요할 수 있다.

모놀로식-마이크로서비스 애플리케이션 간 런타임의 차이점은 다음(표 5)과 같다.

모놀로식 애플리케이션	마이크로서비스 기반 애플리케이션
단일 컴퓨터 노드로 실행된다. 노드는 전체 상태 정보를 모두 알 수 있다.	여러 노드의 집합으로 설계된다. 각 노드는 서비스를 제공한다. 전체 시스템 상태는 개별 노드에서 알 수 없다.
전역 정보 또는 전역 변수를 활용하도록 설계된다.	개별 노드가 로컬에서 사용 가능한 정보를 기반으로 결정을 내린다.
노드의 장애는 애플리케이션의 장애를 의미한다.	한 노드의 장애가 다른 노드에 영향을 주지 않아야 한다.

표 5 : 모놀로식-마이크로서비스 기반 애플리케이션 간 아키텍처의 차이점

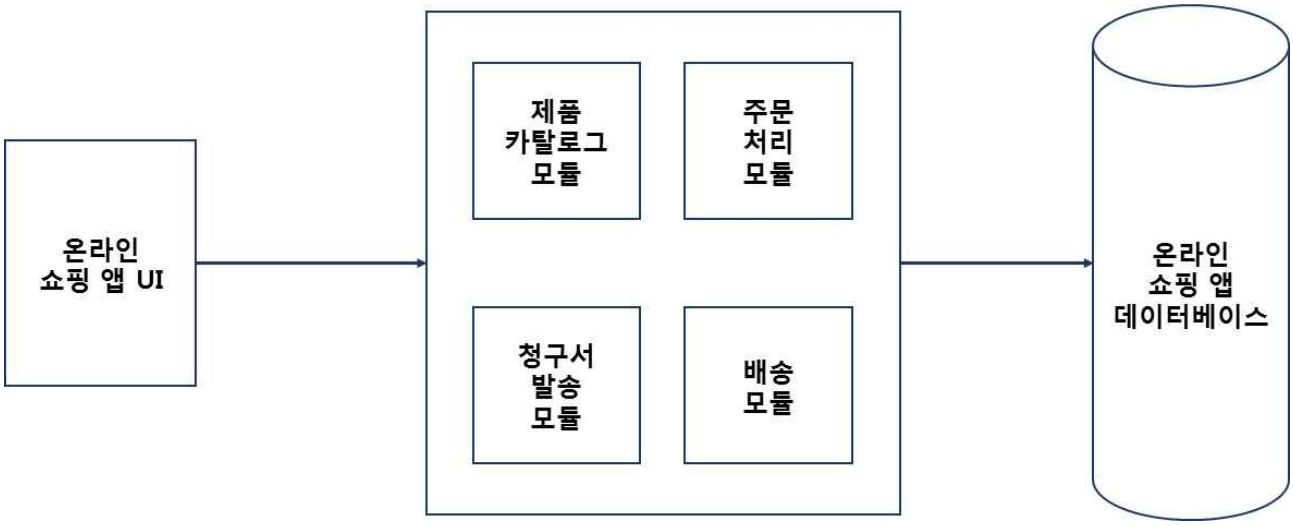

그림 1 : 온라인 쇼핑 애플리케이션 – 모놀로식 아키텍처

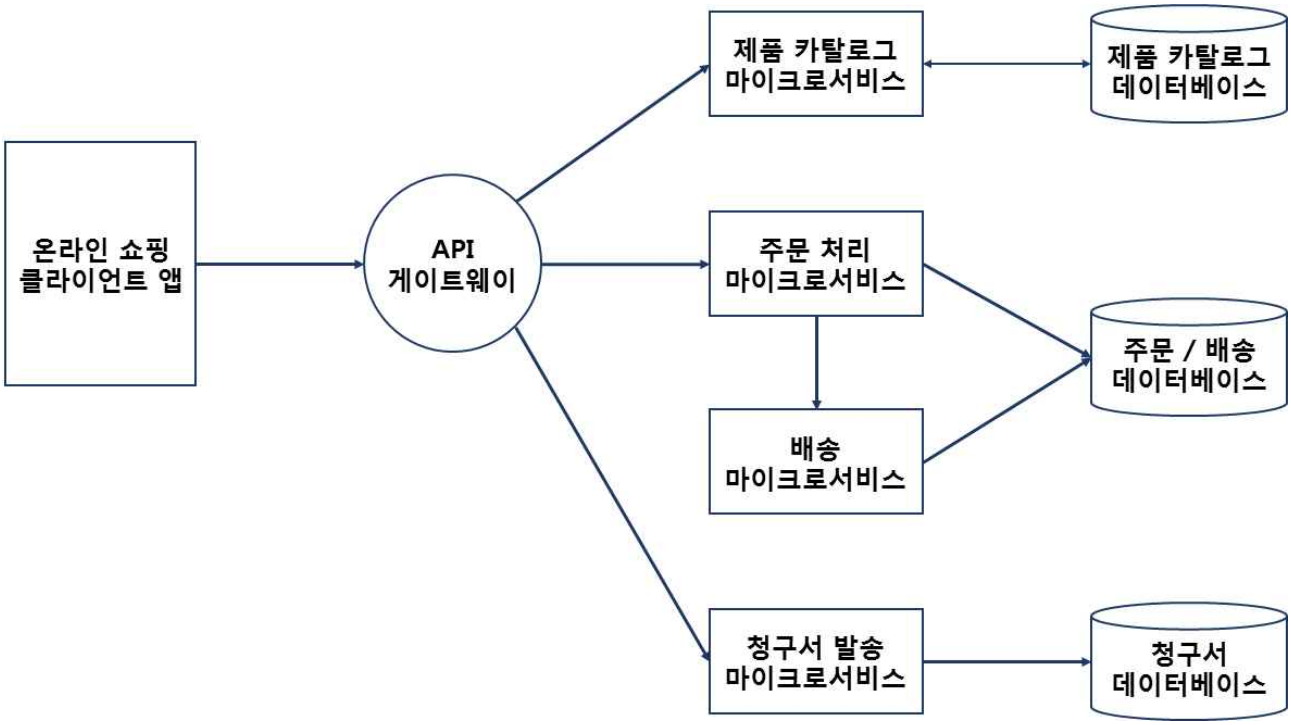

그림 2 : 온라인 쇼핑 애플리케이션 – 마이크로서비스 아키텍처

부록 B. 마이크로서비스 아키텍처 기능에 대한 보안 전략

4장 및 5장에서 논의한 모든 보안 전략(총 13개)을 마이크로서비스의 핵심 기능이나 아키텍처 프레임워크를 명시하여 다음(표 6)에 정리한다.

ID	보안 전략	핵심기능 및 아키텍처 프레임워크
MS-SS-01	• 민감한 데이터에 접근하는 마이크로서비스 API의 경우, 단순히 API 키를 사용하여 인증하는 것은 부적절하다. 중요한 API는 전자 서명(예 : Client Credential Grant) 또는 신뢰할 수 있는 출처에서 검증한 인증 토큰을 통해 접근해야 한다. 추가적으로 토큰 탈취(compromised token)로 인한 피해를 제한하기 위해 일회용 토큰(single-use token) 또는 단기 유효 토큰(short-lived token)을 사용할 수 있다. • 인증 토큰은 핸들(handle) 기반이거나, 암호학적으로 서명하거나, HMAC(Hash-based Method Authentication Code)로 보호되어야 한다. • 애플리케이션에서 사용되는 모든 API 키는 지정된 애플리케이션과 API만 사용할 수 있도록 제한해야 한다. • 모든 API 키의 기능은 신분 증명이 제공하는 보증 수준에 비례하여 범위를 제한해야 한다. • 마이크로서비스 관련 공유 라이브러리를 구현하여 스테이트리스(stateless) 인증 토큰(예 : JWT)을 사용하는 경우, 다음과 같은 보안 예방 조치를 준수해야 한다. (a) 토큰 만료 시간은 가능한 짧아야 한다. 토큰 만료 시간은 세션의 지속 시간을 결정하고, 활성 세션은 취소할 수 없기 때문이다. (b) 토큰 비밀 키를 라이브러리에 하드코딩하지 않아야 한다. 토큰 비밀 키는 동적으로 변경할 수 있어야 한다. 토큰 비밀 키는 데이터 볼트 솔루션에 저장해야 한다. • OAuth 또는 OpenID Connect와 같은 표준 기술을 구현한 경우, 안전하게 배치해야 한다.[19]	인증

ID	보안 전략	핵심기능 및 아키텍처 프레임워크
MS-SS-02	• 모든 API와 리소스에 대한 접근 정책을 정의하고, 인가 서버에서 프로비저닝해야 한다. 개략적인 수준의 접근 정책에는 특정 IP 주소를 가진 기능을 호출할 수 있도록 허용하는 것이 명시된다. 접근 정책은 첫 번째 API 게이트웨이에서 정의되고 시행되어야 한다. 반면, 세부적인 수준의 인가(예 : 특정 마이크로서비스의 비즈니스 로직과 관련된 도메인)는 마이크로서비스의 위치에서 가장 가까운 컴포넌트(예 : 마이크로 게이트웨이) 또는 마이크로서비스 자체에서 정의되고 시행되어야 한다. • 캐싱 메커니즘(Caching Mechanism) : 마이크로서비스에 정책 데이터를 캐시하도록 허용하는 것이 적절할 수 있다. 캐쉬된 정책 데이터는 인가 서버가 가용하지 않은 경우에만 사용해야 하며, 일정 기간이 지난 후 만료시켜야 한다. • 인가 서버는 세부적인 정책을 지원해야 한다. • 인가 서버의 접근 결정은 플랫폼에 중립적인 형식으로 인코딩된 표준화된 토큰(JSON 포맷으로 인코딩된 OAuth 2.0 토큰)을 통해 마이크로서비스(개별/그룹)에 전달해야 한다. 토큰은 핸들 기반 토큰(handle-based token)이나 어설션 기반 토큰(assertion-based token)을 사용할 수 있다. • 내부 인가 토큰(마이크로 게이트웨이가 추가한 토큰 또는 각 요청에 대한 의사결정 지점에서 추가된 토큰)의 범위는 주의 깊게 통제되어야 한다. 예를 들어, 트랜잭션을 요청할 때 인증 토큰에 허용된 범위는 트랜잭션을 완료하기 위해 반드시 접근해야 하는 API 엔드포인트로 한정되어야 한다. • API 게이트웨이에서 모든 다운스트림 마이크로서비스에 대한 인증과 접근 통제를 수행할 수 있다. 즉, 개별 서비스마다 인증과 접근 통제를 제공할 필요가 없어진다. 개별 서비스의 인증과 접근 통제를 구현하지 않은 경우, 네트워크에 특정 위치한 모든 컴포넌트는 API 게이트웨이와 API 게이트웨이의 보호를 우회하여 서비스에 익명으로 연결할 수 있다. 이는 상호 인증과 같은 통제를 통해 서비스에 직접 연결하거나 익명으로 연결되는 것으로 방지함으로써 완화할 수 있다.	접근 관리

ID	보안 전략	핵심기능 및 아키텍처 프레임워크
MS-SS-03	• 전용 서버 또는 서비스 메시 아키텍처의 일부로 서비스 레지스트리 기능을 제공해야 한다. • 서비스 레지스트리 서비스는 가용성과 복원력을 보장하기 위해, QoS(Quality of Service) 설정할 수 있도록 구성된 네트워크에 위치해야 한다. • 애플리케이션 서비스와 서비스 레지스트리 간의 통신은 HTTPS나 TLS(Transport Layer Security)와 같은 안전한 통신 프로토콜을 사용해야 한다. • 서비스 레지스트리는 정당한 서비스만 등록, 갱신, 서비스를 검색하기 위한 데이터베이스 조회를 수행한다는 것을 보장하기 위해 유효성을 검사해야 한다. • 서비스 등록/등록 취소 기능에 대해 마이크로서비스가 독립적으로 서비스되어도 문제가 발생하지 않는 범위(bounded context)와 느슨한 결합 원칙을 준수해야 한다. 즉, 애플리케이션 서비스는 서비스 레지스트리 서비스 등 인프라 서비스와 긴밀하게 결합되는 것을 지양해야 한다. 그리고, 서비스 자체 등록/등록 취소 패턴을 지양해야 한다. 애플리케이션 서비스가 중단되거나 실행 중이지만 요청을 처리할 수 없는 경우, 등록 취소는 전체 프로세스의 무결성에 영향을 미친다. 따라서, 서드 파티 등록 패턴을 사용하여 애플리케이션 서비스의 등록/등록 취소를 수행해야 한다. 그리고, 애플리케이션 서비스는 서비스 레지스트리에 대한 조회만 수행해야 하도록 제한해야 한다. • 서드파티 등록 패턴을 구현한다면, 등록/등록 취소는 애플리케이션 서비스에 대한 헬스 체크가 수행된 후에만 이루어져야 한다. • 대규모 마이크로서비스 애플리케이션의 경우, 분산 서비스 레지스트리를 배치해야 한다. 다수 서비스 레지스트리 인스턴스 간의 데이터 일관성을 유지하기 위해 주의를 기울여야 한다.	서비스 등록 설정

ID	보안 전략	핵심기능 및 아키텍처 프레임워크
MS-SS-04	• 클라이언트가 대상 서비스를 직접 호출하지 않도록 구성해야 하며, 단일 게이트웨이 URL을 가리키도록 구성해야 한다. • 서버-서버 통신뿐만 아니라 클라이언트-API 게이트웨이 통신도 상호 인증 후 수행되어야 하며, 암호화(예 : mTLS 프로토콜)해야 한다. • 상호 작용이 빈번한 서비스는 킵얼라이브 TLS 연결을 생성해야 한다.	안전한 통신
MS-SS-05	• 게이트웨이와 서비스 모두에서 부적절한 행위(예 : 소유 토큰 재사용 공격, 인젝션 공격)를 탐지/경고/대응하기 위해 보안 모니터링을 수행해야 한다. 또한, 입력 검증 오류, 추가 파라미터 오류, 충돌, 코어 덤프를 기록해야 한다. 이러한 기능을 제공하는 소프트웨어로는 OWASP(the Open Web Application Security Project) 앱센서(AppSensor)가 있다. 앱센서는 게이트웨이, 서비스 메시, 마이크로서비스에 구현될 수 있다. • 중앙 대시보드는 다양한 서비스와 서비스를 연결하는 네트워크 세그먼트의 상태를 표시한다. 최소한 대시보드는 입력 검증 실패, 인젝션 공격 시도가 명확한 파라미터와 같은 보안 파라미터를 보여주어야 한다. • 비즈니스 로직의 의사결정 결과, 접근 시도 등의 관점에서 정상적이고 위험하지 않은 행위의 기준을 만들어야 한다. IDS(Intrusion Detection System)는 이 기준을 위반하는 행위를 탐지할 수 있도록 충분한 기능을 보유해야 하며, 적절한 위치에 배치해야 한다.	보안 모니터링
MS-SS-06	• 신뢰할 수 있는 컴포넌트를 프락시만으로 제한하기 위해 프락시 서킷브레이커 옵션을 적용해야 한다. 프락시 서킷브레이커 옵션은 클라이언트와 마이크로서비스에 신뢰(예 : 임계값을 설정하고 설정된 임계값을 기준으로 요청을 차단)를 부여할 필요가 없다. 클라이언트와 마이크로서비스는 다수가 존재하기 때문이다.	서킷브레이커 구현

ID	보안 전략	핵심기능 및 아키텍처 프레임워크
MS-SS-07	• 부하 분산 기능을 지원하는 모든 프로그램을 개별적인 서비스 요청으로부터 분리해야 한다. 예를 들어, 서비스에서 부하 분산 풀(pool)을 결정하기 위해 헬스 체크를 수행하는 프로그램은 백그라운드에서 비동기로 실행해야 한다. • 로드 밸런서와 마이크로서비스 플랫폼 사이의 네트워크 연결을 보호하기 위해 주의를 기울여야 한다. • DNS 리졸버가 마이크로서비스 앞에 배치되어 가용한 마이크로서비스 인스턴스의 테이블을 제공하는 경우(DNS 리졸버를 사용한 서버 사이드 서비스 검색 패턴), DNS 리졸버는 단일 목록을 호출한 마이크로서비스에게 표시하기 위해 헬스체크 프로그램과 동시에 작동해야 한다.	부하분산 구현
MS-SS-08	• 애플리케이션 사용에 대한 할당량 또는 제한은 인프라 및 애플리케이션 요구사항 모두에 기반해야 한다. • 명확한 API 사용 계획에 기반하여 제한을 결정해야 한다. • 높은 보안이 요구되는 마이크로서비스는 재사용 공격 탐지(replay detection) 기능을 구현해야 한다. 위험을 기반으로 항상 재사용 공격을 탐지하거나, 임의로 탐지하도록 설정할 수 있다.	사용량 제한 (스로틀링)
MS-SS-09	• 서비스의 기존 버전과 새로운 버전에 대한 트래픽은 통제된 방법으로 새로운 버전으로 전환되는지 모니터링하고, 카나리 릴리즈와 관련한 위험을 모니터링 하기 위해 API 게이트웨이와 같은 중앙 노드를 통해 전달되어야 한다. 기존 버전과 최신 버전을 호스팅하는 노드 모두에 대한 보안을 모니터링해야 한다. • 기존 버전에서 새로운 버전으로 트래픽을 지속 증가시켜야 한다. • 새로운 버전의 성능과 기능 정확도는 새로운 버전으로 트래픽을 증가시키는 요소여야 한다. • 카나리 릴리즈를 설계할 때, 기존 버전 또는 새로운 버전에 대한 클라이언트의 선호도를 고려해야 한다.	새로운 버전의 마이크로서비스 반영

ID	보안 전략	핵심기능 및 아키텍처 프레임워크
MS-SS-10	• 클라이언트에 대한 세션 정보를 안전하게 저장해야 한다. • 바인딩된 서버 정보를 전달하는데 사용되는 아티팩트(세션 쿠키)를 보호해야 한다. • 내부 인가 토큰을 사용자에게 제공하지 않아야 한다. 또한, 사용자의 세션 토큰을 정책 결정에 사용하기 위한 목적으로 게이트웨이를 넘겨 전달하지 않아야 한다.	세션 지속 처리
MS-SS-11	• 크리덴셜 도용에 대한 예방 전략은 오프라인 전략보다 런타임 전략이 효과적이다. 특정 위치(예 : IP 주소)에서 지정된 시간 간격 동안 로그인 시도 횟수에 대한 임계값을 설정해야 한다. 임계값을 초과하면, 인증/인가 서버는 예방을 위한 조치를 실행해야 한다. 소유 토큰(bearer token)을 사용하는 경우, 토큰 재사용의 예방 및 탐지를 위해 이 기능이 반드시 존재해야 한다. • 크리덴셜 스터핑 탐지 솔루션은 탈취된 크리덴셜 데이터베이스에 대비하여 사용자 로그인을 검사하고, 정당한 사용자에게 크리덴셜이 탈취되었음을 경고하는 기능을 보유해야 한다. • DoS 공격을 탐지하고, 서비스가 접근이 불가해지기 전에 경고하도록 IDS와 네트워크의 경계에 위치한 장비(boundary device)를 구성해야 한다. • 악성코드로 인한 위협을 완화하기 위해 파일 업로드, 컨테이너의 메모리 및 파일 시스템을 검사하도록 서비스 호스트를 구성해야 한다.	크리덴셜 도용 및 스터핑 공격 예방

ID	보안 전략	핵심기능 및 아키텍처 프레임워크
MS-SS-12	• API 게이트웨이와 계정 관리 애플리케이션을 통합하여 크리덴셜을 제공한 후, API를 활성화해야 한다. • API 게이트웨이를 통해 계정 관리 애플리케이션을 호출할 경우, ID 제공자(IdP, identity provider)와 통합하기 위한 커넥터가 제공되어야 한다. • API 게이트웨이는 클라이언트의 요청에 대해 액세스 토큰을 생성할 수 있는 아티팩트(예 : OAuth 2.0 인가 서버)에 대한 커넥터를 보유해야 한다. • 공격(예 : DoS, 악의적인 행위)을 탐지하고 성능 저하의 원인을 밝혀내기 위해 모든 트래픽 정보를 모니터링 및 분석 애플리케이션으로 안전하게 전송해야 한다. • 분산 게이트웨이를 배치하는 경우, 게이트웨이 간 토큰을 변환(교환)하는 서비스가 존재해야 한다.[18] 첫 번째 게이트웨이에서 제공하는 토큰은 광범위한 사용 권한을 가져야 한다. 반면, 내부 게이트웨이(또는 마이크로 게이트웨이)에서 제공하는 토큰은 범위가 좁고, 특정한 권한을 가져야 한다. 또는 대상 마이크로서비스 플랫폼에 적합한 완전히 다른 형태의 토큰이어야 한다. 이는 최소 권한의 법칙을 구현하는데 도움을 준다.	API 게이트웨어 설정

ID	보안 전략	핵심기능 및 아키텍처 프레임워크
MS-SS-13	• 서비스 간 통신에 사용할 프로토콜을 지정하고, 애플리케이션의 요구사항에 따라 서비스 상이의 트래픽 부하를 지정하기 위한 정책을 지원해야 한다. • 모든 서비스에 대해 접근 통제 정책을 항상 실행하도록 기본 설정되어야 한다. • 권한 상승을 초래할 수 있는 설정(예 : 서비스 역할 권한, 서비스 사용자 계정에 대한 서비스 역할 바인딩)은 지양해야 한다. • 서비스 메시는 컴포넌트의 리소스 사용 한계를 지정할 수 있는 기능이 있어야 `한다. 이 기능이 없으면, 컴포넌트가 전체 마이크로서비스 애플리케이션의 복원력과 가용성에 영향을 미칠 수 있다. • 서비스 메시는 환경 분석 결과(요청된 분석 결과 포함)를 수집하고, 모니터링을 위해 중앙화된 서비스로 전송하도록 설정할 수 있는 기능이 있어야 한다. 정책은 멀티 클러스터 마이크로서비스 환경에 대한 고가용성과 복원력을 보장하기 위해 싱글 서비스 메시 또는 멀티 서비스 메시(각 서비스 메시는 자체 컨트롤 플레인을 가짐)를 지정하는 것을 감안해야 한다. • 중요도가 높은 마이크로서비스 기반 애플리케이션이라면, 오케스트레이션 플랫폼 내부에서 서브넷팅(subnetting)으로 네트워크를 분할(segmentation)해야 한다. 서비스 메시 레이어는 전체적으로 세션 제한으로 네트워크를 분할한다. 서브넷팅은 사이드카 프록시(네트워크 트래픽을 보호/차단하기 위해 사용)를 우회하는 악의적인 행위자에 의한 위협에 대응하기 위한 보완책이다.	서비스 메시 설정

부록 C. 참고 문헌

[1] Sill A (2016) The design and architecture of microservices. IEEE Cloud Computing 3(5):76-80. https://doi.org/10.1109/MCC.2016.111

[2] Richardson C, Smith F (2016) Microservices: From design to deployment (NGINX Inc.). Available at https://www.nginx.com/resources/library/designing-deployingmicroservices/

[3] TechTarget (n.d.) Comparing two schools of application development: Traditional vs. Cloud-Native. Available at https://searchcloudcomputing.techtarget.com/PaaS/Comparing-Two-Schools-ofApplication-Development-Traditional-vs-Cloud-Native

[4] Richardson C (2015) Building microservices: Using an API gateway. Available at https://www.nginx.com/blog/building-microservices-using-an-api-gateway/

[5] Palladino M (2016) Microservices and API gateway, Part 1: Why an API gateway? Available at https://shadrin.org/nginx/blog/content/microservices-api-gateways-part-1why-an-api-gateway.html

[6] Jander K, Braubach L, Pokahr A (2018) Defense in-depth and role authentication for microservice systems. Procedia Computer Science 130:456-463. https://doi.org/10.1016/j.procs.2018.04.047

[7] Richardson C (2015) Building microservices: Inter-process communication in a microservices architecture. Available at https://www.nginx.com/blog/buildingmicroservices-inter-process-communication/

[8] Harms H, Rogowski C, Lo Iacono L (2017) Guidelines for adopting frontend architectures and patterns in microservices-based systems. Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ACM, Paderborn, Germany), pp 902-907. https://doi.org/10.1145/3106237.3117775

[9] Montesi F, Weber J (2016) Circuit breakers, discovery, and API gateways in microservices. arXiv preprint. https://arxiv.org/abs/1609.05830v2

[10] O'Neill M, Malinverno P (2018) Critical capabilities for full life cycle API management. (Gartner, Stamford, CT), ID G00334223. Available at https://www.gartner.com/doc/reprints?id=1-51SE2EK&ct=180601&st=sb

[11] Calcote L (2018) The enterprise path to service mesh architectures (O'Reilly Media, Sebastopol, CA).

[12] Twistlock (n.d.) Securing the service mesh: Understanding the value of service meshes, why Istio is rising in popularity, and exploring official Twistlock compliance checks for Istio. Available at https://www.twistlock.com/resources/securing-service-mesh-istiocompliance-checks/

[13] Yarygina T, Bagge, AH (2018). Overcoming security challenges in microservice architecture. Proceedings of 2018 IEEE Symposium on Service-Oriented System Engineering (IEEE, Bamberg, Germany), pp 11-20. https://doi.org/10.1109/SOSE.2018.00011

[14] OpenID (2019) Welcome to OpenID Connect. Available at https://openid.net/connect/

[15] Hardt D (ed.) (2012) The OAuth 2.0 authorization framework. (Internet Engineering Task Force), IETF Request for Comments (RFC) 6749. https://doi.org/10.17487/RFC6749

[16] NGINX (n.d.) High-performance load balancing: Scale out your applications with NGINX and NGINX Plus. Available at https://www.nginx.com/products/nginx/loadbalancing/

[17] Katz O (2017) Improving credential abuse threat mitigation. Available at https://blogs.akamai.com/2017/01/improving-credential-abuse-threat-mitigation.html

[18] Jones M, Nadalin A, Campbell B (ed.), Bradley J, Mortimore C (2018) OAuth 2.0 token exchange. (Internet Engineering Task Force), IETF Internet-Draft. Available at https://datatracker.ietf.org/doc/draft-ietf-oauth-token-exchange/

[19] Lodderstedt T, Bradley J, Labunets A, Fett D (2019) OAuth 2.0 security best current practice. (Internet Engineering Task Force), IETF Internet-Draft. Available at https://datatracker.ietf.org/doc/draft-ietf-oauth-security-topics/

[20] Jain J (2015) HTTP verb tempering: Bypassing web authentication and authorization. Available at https://resources.infosecinstitute.com/http-verb-tempering-bypassing-webauthentication-and-authorization/