AI 시대의 보안: 복잡성에서 맥락으로

Security Tech Leader

hellsonic / 이종호

기술 발전과 복잡성의 등장

서비스 구조의 급격한 복잡성 증가 — MSA, 클라우드, 자동화, AI

아키텍처의 변화

- 수십·수백 개의 독립적인 서비스로 변화
- 각 서비스 간의 통신과 의존성이 기하급수적으로 증가

인프라의 복잡성

- 멀티 클라우드와 하이브리드 환경의 확산
- 컨테이너 오케스트레이션의 도입 -> 인프라 경계 유동적으로 변화

자동화의 가속화

- CI/CD, IaC, GitOps 등 자동화 도구 -> 배포 속도 비약적으로 향상
- 개발·운영 과정의 각 단계마다 자동화된 프로세스와 관리 포인트 증가

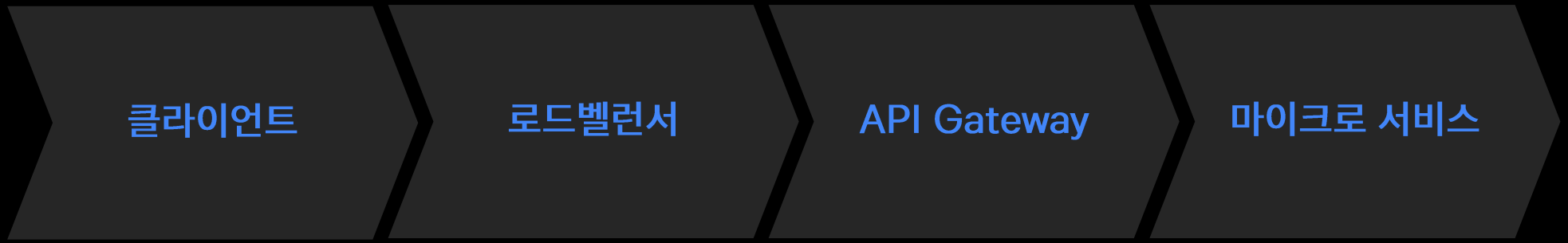
AI 통합에 따른 새로운 복잡성

- 추천 엔진, 이상 탐지, 자동화된 의사결정 등 AI 기술이 시스템 곳곳 에 통합
- 운영 효율성은 높아졌지만 동시에 새로운 복잡성 레이어 추가

toss

기술 발전과 복잡성의 등장

1개의 요청 -> 10~20개의 컴포넌트 통과



- API 호출 1개 당 여러 개의 마이크로 서비스, 캐시, 큐, 데이터베이스 거침
- 각 홉(hop) 마다 인증, 권한, 로깅, 에러 처리 방식이 제각각 적용 됨
- 인프라, 서비스 계층별 일관성 확보가 어려워짐

기술 발전과 복잡성의 등장

AI 통합 -> 새로운 복잡성 -> 새로운 공격 포인트 등장

Prompt Injection	악의적인 프롬프트를 통한 모델 행동 조작
Model Poisoning	학습 데이터 오염을 통한 모델 변조
Data Leakage	모델 출력을 통한 민감 정보 유출
Context Overflow	긴 컨텍스트를 통한 메모리/성능 공격

기술 발전과 복잡성의 등장

클라우드, API, 모델이 섞이며 사라진 "보호 대상" 경계

전통적 자산 정의

서버 IP 목록
네트워크 세그먼트
애플리케이션 목록
데이터베이스

현대적 자산 정의

데이터 흐름과 변환 과정
서비스 간 신뢰 관계
인증 상태와 권한 경로
API 엔드포인트

코드, 데이터, 모델, 프로세스가 함께 얽혀 '자산 경계' 자체가 흐려진 구조
정적인 인벤토리만으로는 부족 -> 동적인 관계망 관점 필요

toss

복잡성이 만드는 새로운 보안 패러다임

코드 중심 → 프로세스 중심 전환

과거의 접근 방식

코드 스니펫만 분석하고, HTTP Request와 Response의 직접적인 흐름만 추적

- 입력 값 검증 우회
- SQL Injection, XSS
- 단일 지점의 취약점

현재의 접근 방식

전체 프로세스를 파악하고 전략적 타겟을 선정

- 시스템 간 연결고리 분석
- 프로세스 간 취약점
- 복잡성에 따른 신규 위협

프로세스에 대한 깊은 이해의 중요성

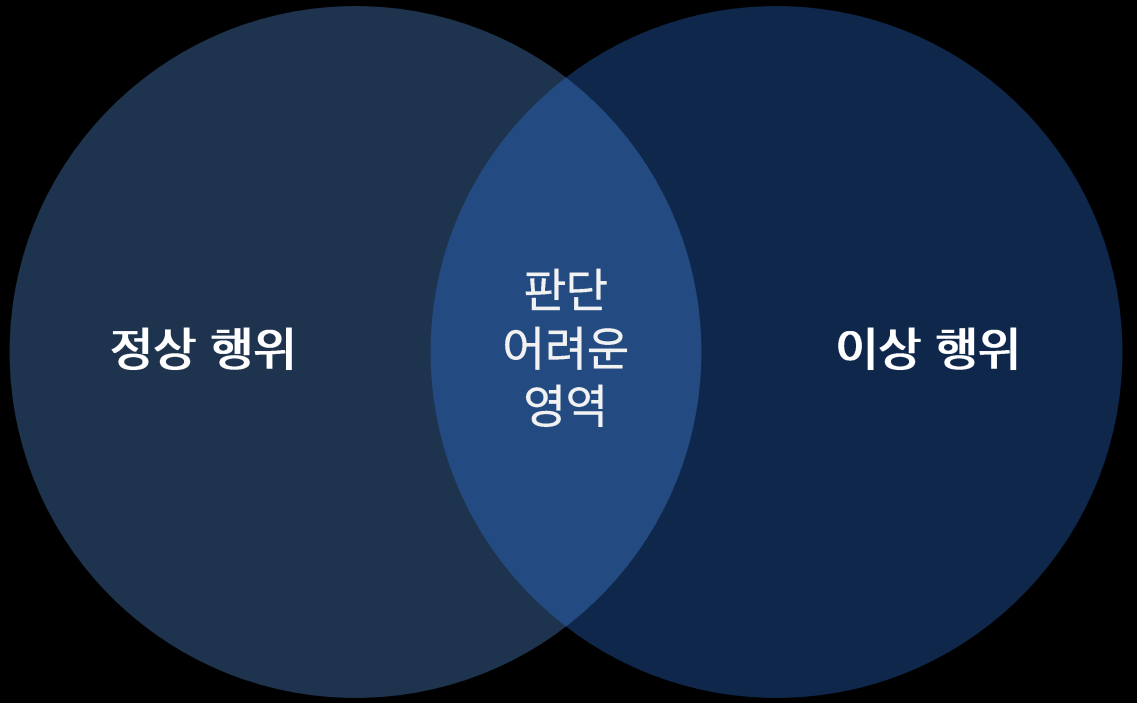
실제 영향 있는 취약점을 발견하기 위해서는 **전체 여정**을 이해하는 것이 중요.

단일 코드 분석을 넘어서 데이터가 흐르는 모든 경로와 상호작용하는 모든 시스템을 파악하면 새로운 공격대상을 발견할 수 있음

toss

복잡성이 만드는 새로운 보안 패러다임

이상행위 모니터링 탐지 관점에서도 높아지는 난이도



ex. 2FA 로그인은 모두 정상인데, 30초 사이 서로 다른 국가에서 토큰 재발급이 반복되는 패턴

toss

복잡성이 만드는 새로운 보안 패러다임

단일 신호로 이상행위 탐지 불가능

단일 이벤트 관점

"단일 이벤트만으론 판단할 수 없다."

✓ 로그인 성공

올바른 자격증명

✓ API 호출

정상적인 엔드포인트

✓ 데이터 조회

권한 내 리소스

각각은 완벽하게 정상

패턴 결합 관점

하지만 조합하면?

⚠ 비정상 패턴

10초 내 100회 로그인

⚠ 의심스러운 순서

인증 없이 민감 API 접근

⚠ 이상 행동

평소와 다른 데이터 패턴

개별 이벤트는 정상이어도, 조합 패턴이 비정상을 드러냄

toss

복잡성이 만드는 새로운 보안 패러다임

결국 맥락(Context)를 파악해야 정상 vs 이상 구분 가능

Context

정상 vs 이상

판정을 위한 맥락

관계

✓ 서비스와 데이터 흐름의 연결

순서

✓ 작업의 선후
✓ 작업의 선행 조건

상태

✓ 현재 시스템 및 이전 결과의 지속성

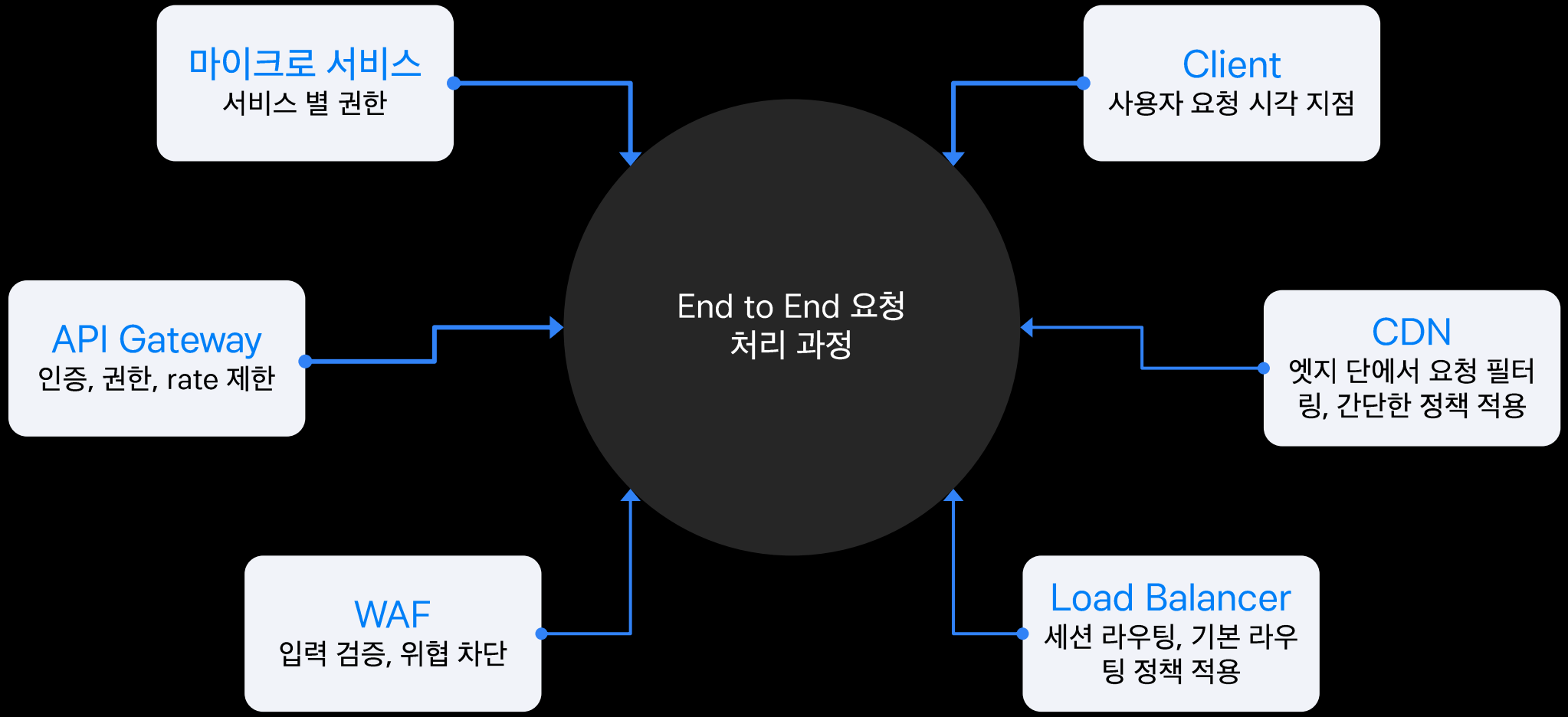
시간

✓ 작업 시점, 세션과 타임아웃

toss

복잡성이 만드는 새로운 보안 패러다임

단일 함수가 아니라 전체 프로세스를 봐야 함



toss

서비스 취약점의 어려운점: '자산 상태 불일치'

코드 취약점의 상대적 비중 감소 요인

도구의 진화

- ✓ SAST/DAST/IAST 기술의 탐지 정확도와 커버리지가 지속적으로 향상됨

AI 개발 환경

- ✓ AI를 통한 코드 리뷰와 자동 수정의 일상화
- ✓ 보안 품질 지속적 개선 효과

표준 준수

- ✓ 프레임워크 내장 보안 기능 강화
- ✓ 보안 코딩 표준 준수 확대

문화 확산

- ✓ 개발자의 보안 인식 개선
- ✓ 보안 중심 코드 작성 문화 정착

toss

서비스 취약점의 어려운점: '자산 상태 불일치'

체인화 된 취약점 악용

1단계: 인증 우회

낮은 수준의 인증으로 초기 접근 획득

- ✓ 2FA 로그인은 강한 인증 / 비밀번호 재설정은 약한 인증
  → 인증 강도 불일치

2단계: 권한 확대

서비스 간 신뢰 관계 악용

- ✓ 이메일, 세션 기반 인증 요소가 조합될 때 예상치 못한 권한 상승 가능

3단계: 데이터 탈취

높은 권한으로 민감 데이터 접근

- ✓ 관리 권한으로 내부 데이터 접근 및 다량의 유출

toss

서비스 취약점의 어려운점: '자산 상태 불일치'

SAST/DAST 취약점 점검 도구의 한계

코드 중심 Audit

```
function transferMoney(from, to, amount) {
        if (!isAuthenticated(from))
                return;
        if (amount <= 0)
                return;
        if (getBalance(from) < amount)
                 return;
        deduct(from, amount);
        credit(to, amount);
}
```

- 코드만 보면 안전해 보임
- 각 검증이 제자리에 있고, SQL 인젝션 가능성 없음

흐름 관점의 문제

실제 시스템

- getBalance()와 deduct() 사이에 다른 서비스 호출 가능
- 잔액 조회와 차감 시간 차를 악용하면 이중 지불 가능
- 다른 이벤트가 끼어들 수 있어 일관성 보장이 어려움

toss

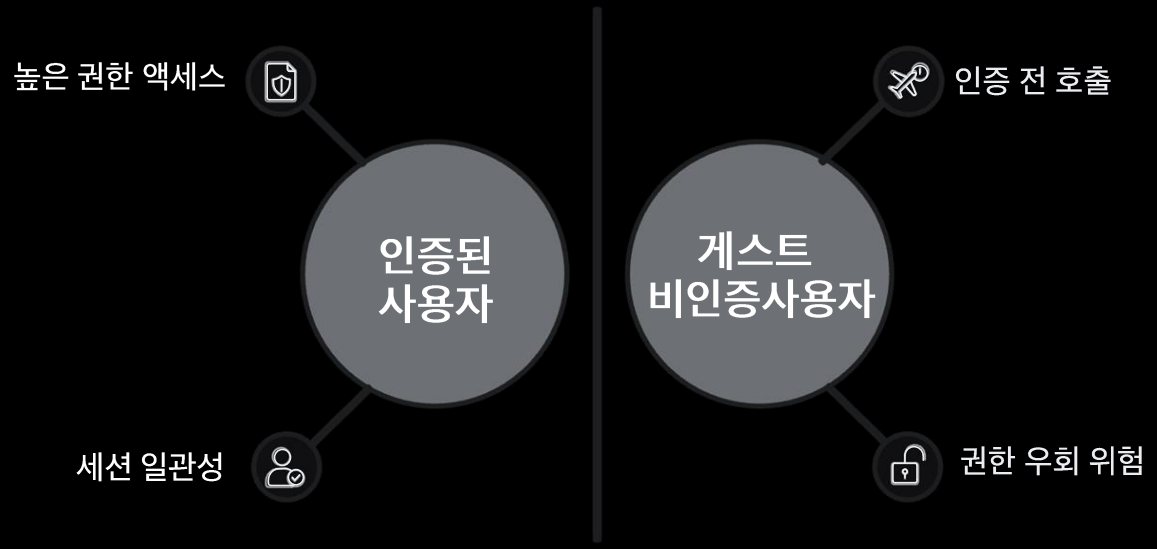
서비스 취약점의 어려운점: '자산 상태 불일치'

동일한 코드여도 맥락, 프로세스의 차이로 발생하는 취약점 多

동일한 함수 호출이라도:

- 누가 호출하는가?
- 어디서 호출하는가?
- 언제 호출하는가?
- 어떤 상태에서 호출하는가?

맥락, 프로세스에 따른 보안 평가 진행



예시

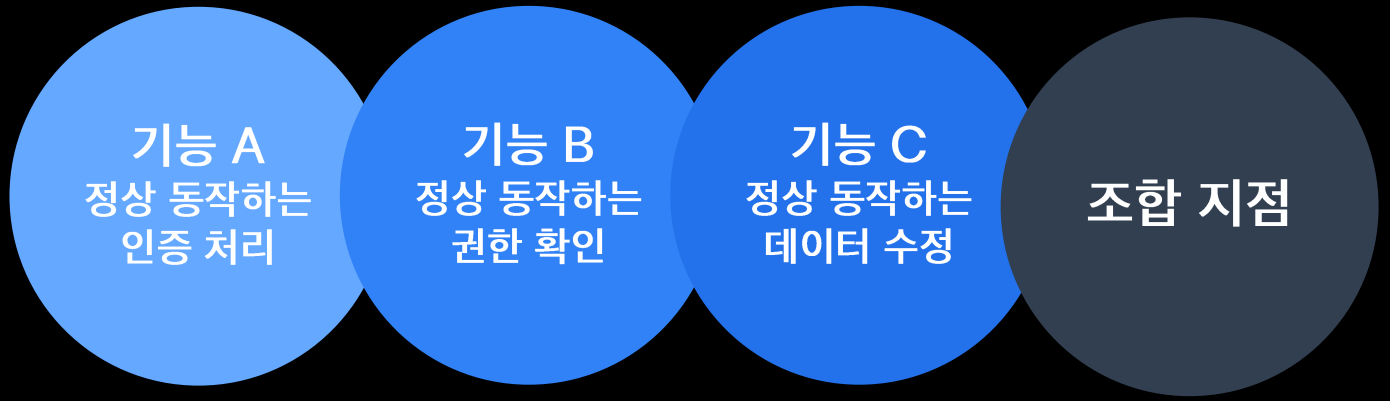
✓ updateUserProfile() 함수는 코드 상 안전

✓ 이메일 인증 전에 호출되면? 다른 사용자의 세션에서 호출되면? 관리자 권한으로 우회되면?

toss

서비스 취약점의 어려운점: '자산 상태 불일치'

기능은 정상이지만, 비즈니스 로직이 조합된 결과 위험한 상태



인증 강도 불일치

로그인은 2FA 필요, 비밀번호 재설정은 이메일만

권한 상승 경로

일반 기능의 조합으로 관리자 권한 획득

Race Condition

동시 요청으로 일관성 검증 우회

상태 전이 오류

중간 상태에서의 예상치 못한 권한

toss

자산 정형화 + AI 맥락 분석

자산의 흐름의 구조화를 통한 복잡성 관리

코드 레벨 분석의 한계

개별 취약점 탐지에 집중

시스템 레벨 리스크

구조적 취약점은 간과

정형화를 통한 해결

정형화된 명세 기반으로 AI가 맥락적 추론을 수행할 수 있음

복잡성은 피할 수 없지만, 구조화를 통해 통제력을 확보할 수 있음

toss

자산 정형화 + AI 맥락 분석

자산의 흐름의 구조화를 통한 복잡성 관리

시스템 흐름 명시화
요청부터 응답까지 전체 경로
구조화 하여 표현

↓

숨겨진 취약점 도출

상태와 권한 체계화
각 엔드포인트 인증 레벨과
상태 전이 규칙 명확하게 정의

↓

논리적 불일치 사전 식별

데이터 관계 구조화
Entity 간 의존성과
참조 관계 명시적 표현

↓

데이터 흐름 무결성 검증

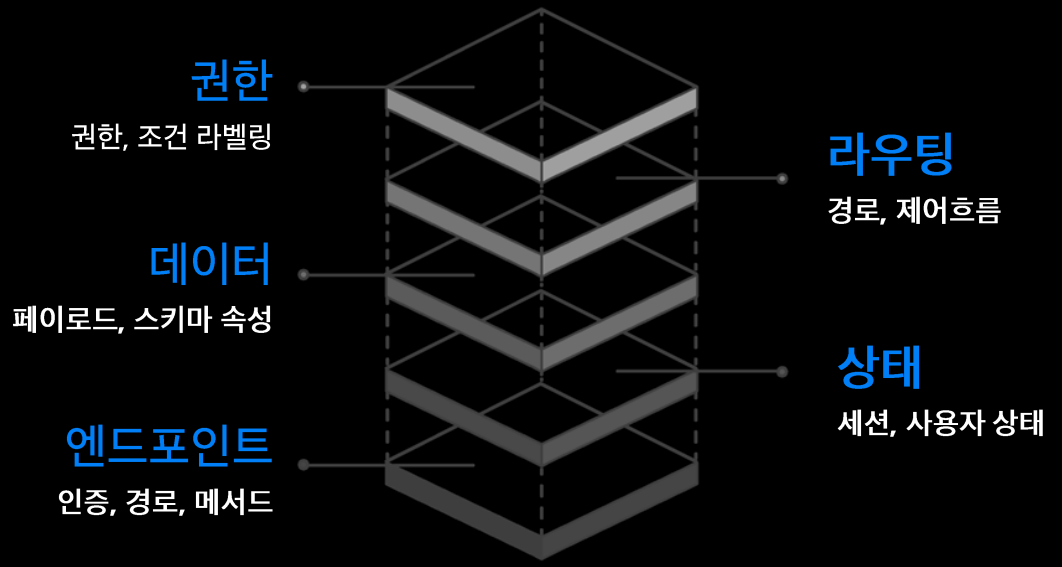
toss

자산 정형화 + AI 맥락 분석

자산의 흐름의 구조화를 통한 복잡성 관리

정형화의 핵심 요소

1. **자산정의** — 시스템 구성요소 명확히 식별 후 분류
   엔드포인트, 데이터, 상태, 권한
2. **관계 명세** — 자산 간 의존성과 상호작용 정의
   데이터 흐름, 제어 흐름, 신뢰관계
3. **속성 표준화** — 보안 관련 속성의 일관된 표현
   인증 강도, 권한 레벨, 민감도 등급
4. **규칙 체계화** — 보안 정책과 제약사항의 구조화
   허용/금지 조건, 검증 요구사항, 예외 상황



수학적 엄밀함을 추구할 필요 없이, 누구나 쉽게 작성 가능한 구조화된 표현으로
LLM이 맥락을 추론해 취약한 패턴을 발견할 수 있음

toss

자산 정형화 + AI 맥락 분석

Case #1

간단한 인증 페이지에서의 자산 정의

Path	Input	Output	인증	주요기능 / 영향
/api/login	id, password, 2fa_code	Session	None	로그인 성공 시 Session 생성
/api/profile	userId	User_Profile	Session	사용자 정보 조회

- ✓ 사용자 정보를 반환하는 internal api인 /internal/profile 을 외부 /api/profile 에 매핑
- ✓ 로그인 이후 접근할 수 있는 /api/profile API가 현재 Session에 해당하는 사용자 정보 조회가 아닌, userId 를 참조

toss

자산 정형화 + AI 맥락 분석

규칙 없이도, 자산 정의만으로 프로세스의 취약점들을 찾아낼 수 있음

```yaml
system_assets:
  endpoints:
    - id: login
      path: /api/login
      method: POST
      auth:
        type: none
      params:
        - name: id
          type: string
          required: true
        - name: password
          type: string
          required: true
        - name: 2fa_code
          type: string
          required: false
      result: session_created
      description: 사용자 로그인 성공 시 세션 생성

    - id: get_profile
      path: /api/profile
      method: POST
      auth:
        type: session
      params:
        - name: userId
          source: body
          required: true
      result: profile
      description: 로그인 사용자의 프로필 정보 조회
```

Invariant 또는 명시적 규칙 없이
단순히 자산(path, input, output, 설명)을 YAML로 작성

1. Insecure Direct Object Reference (IDOR)

```
POST /api/profile
Authorization: [valid session]
{
  "userId": 1234  // 다른 사용자의 ID
}
```

- **공격 방법:** 자신의 유효한 세션으로 다른 사용자의 userId를 요청
- **결과:** 타인의 프로필 정보 무단 열람
- **즉시 실행 가능:** 세션만 있으면 모든 사용자 정보 접근 가능

toss

자산 정형화 + AI 맥락 분석

Case #2

간단한 인증 페이지에서의 자산 정의

Path	Input	Output	인증	주요기능 / 영향
/api/login	id, password, 2fa_code	Session	None	로그인 성공 시 Session 생성
/api/password-reset	email_token	Session	None	패스워드 변경 후 자동 로그인

✓ 패스워드 찾기 후 패스워드를 변경시키고 사용자 편의성을 위해 자동로그인

toss

자산 정형화 + AI 맥락 분석

규칙 없이도, 자산 정의만으로 프로세스의 취약점들을 찾아낼 수 있음

```yaml
system_assets:
  endpoints:
    - id: login
      path: /api/login
      method: POST
      auth:
        type: none
      params:
        - name: id
          type: string
          required: true
        - name: password
          type: string
          required: true
        - name: 2fa_code
          type: string
          required: true
      result: session_created
      description: 사용자 로그인 성공 시 세션 생성

    - id: password_reset
      path: /api/password-reset
      method: POST
      auth:
        type: none
      params:
        - name: email_token
          type: string(256bytes random code)
          required: true
      result: session_created
      description: 이메일 토큰을 통해 비밀번호를 재설정하고 자동 로그인
```

핵심 취약점: 2FA 완전 우회

이메일 토큰으로 2FA 우회

```
POST /api/password-reset
{
  "email_token": "valid_256byte_token"
}
```

- **공격 시나리오:**
  1. 피해자 이메일에 패스워드 리셋 요청
  2. 이메일 해킹하거나 이메일 계정 탈취
  3. 리셋 토큰으로 **2FA 없이** 즉시 로그인 세션 획득
- **실제 피해: 모든 2FA 보안 조치 무력화**

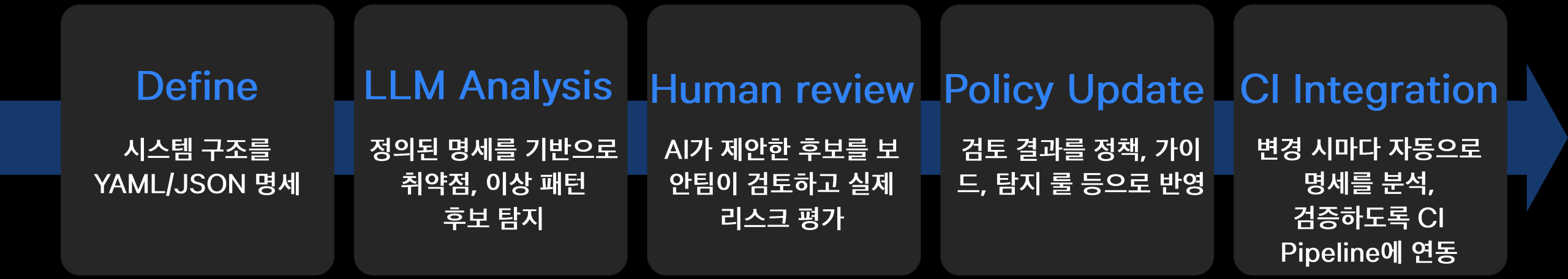
이메일 = 단일 인증 요소

- 로그인: `ID + 패스워드 + 2FA` (3요소)
- 패스워드 리셋: `이메일 토큰만` (1요소)
- **결과:** 이메일만 탈취하면 모든 계정 접근 가능

toss

자산 정형화 + AI 맥락 분석

정형화된 명세에서 LLM을 활용한 취약점 탐색



toss

자산 정형화 + AI 맥락 분석

정형화된 명세에서 LLM을 활용한 취약점 탐색

자동화 할 영역

- ✓ 코드 내 엔드포인트 자동 추출
- ✓ 스키마 일관성 검증
- ✓ LLM 을 통한 패턴 기반 탐지
- ✓ 이상 징후 리포트 생성

반복적이고 확장 가능한 작업 -> "자동화"

시스템 변경 시 마다 자동 실행

사람의 판단 영역

- ✓ 시스템 맥락과 비즈니스 로직 이해
- ✓ LLM이 찾은 패턴의 실제 위험도 평가
- ✓ False positive 필터링
- ✓ 보안 정책 의사결정

맥락적 이해 전략적 의사결정은

여전히 사람의 판단 영역

toss

자산 정형화 + AI 맥락 분석

정형화된 명세에서 LLM을 활용한 취약점 탐색 장점

패턴 인식 능력

- 복잡한 상호작용에서 취약점 발견
- 유사 사례 학습 -> 새로운 취약점 예측
- 다양한 공격 벡터 조합 분석

명세 분석 능력

- 불완전하거나 모순된 명세 검출
- 암묵적 가정이나 누락된 요구사항 식별
- 크로스 레퍼런스 분석으로 일관성 검증

창의적 공격 시나리오 도출

- 기존 도구가 놓치는 논리적 취약점 발견
- 다단계 공격 체인 구성
- 비즈니스 컨텍스트를 고려한 실질적 위험 평가

toss

자산 정형화 + AI 맥락 분석

정형화된 명세에서 LLM을 활용한 취약점 탐색의 문제점

정형화와 AI 활용의 현재 시점 문제

- ✓ False Positive 과다 발생
- ✓ 명세 작성 및 유지보수 부담
- ✓ 도구 성숙도 부족
- ✓ 비용 대비 효과 의문

2년 후의 AI는 지금과 완전히 다른 차원 예상

- ✓ AI 발전속도 기하급수적 성능 향상
- ✓ OpenAI, Claude, Gemini, Grok 치열한 경쟁
- ✓ 코드 이해, 생성 능력, 추론모델들의 폭발적 성장
- ✓ 멀티 모달 능력 : 텍스트 + 코드 + 시스템 구조 통합 분석

toss

결론

코드에서 시스템으로 확장

코드 취약점 감소
시스템 구조를
YAML/JSON 정의

정형화 발전
시스템 전체 AI가 이해

보안 평가 진화
코드 점검을 넘어
시스템으로 확장

통제 가능성 확보
구조화를 통해
복잡한 서비스 관리

toss

결론

AI 시대 보안의 새로운 접근 방향

	전통적 초점	확장된 초점
보안 대상	코드 내 취약점	자산 상태와 맥락
분석 단위	주요 함수	프로세스와 시스템 전체
도구	취약점 탐지 중심	구조적 이해 중심
보안의 역할	점검 중심	설계 및 운영 단계의 통제 중심

toss

감사합니다