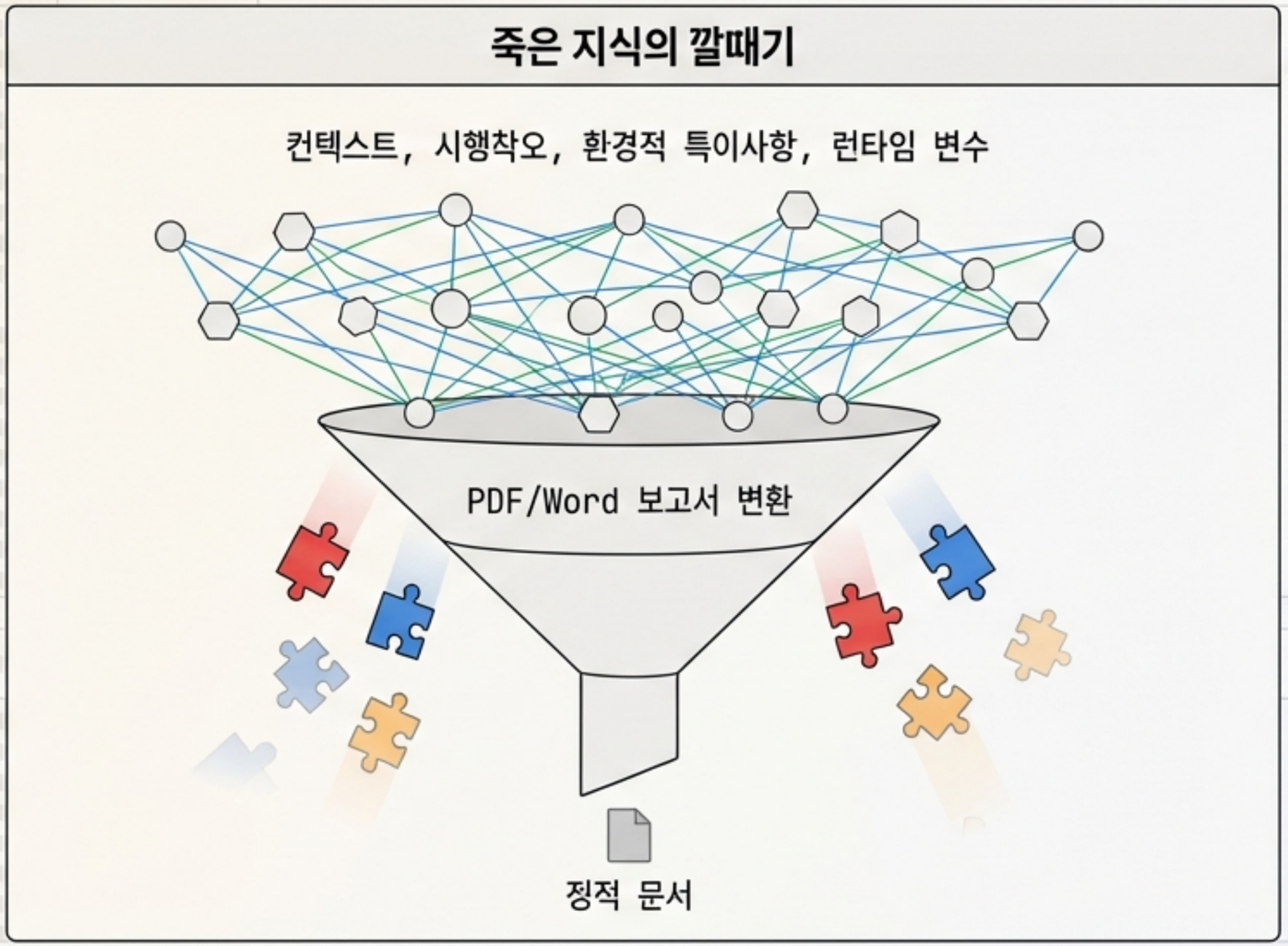
보안 진단 보고서는 발행되는 순간 죽는다

[PUBLISHED & OBSOLETE]

단절된 정적 문서에서 살아있는 코드로:
Security Testing as Code (STaC) 도입을 위한 청사진

결과만 남고 맥락은 증발하는 진단 프로세스



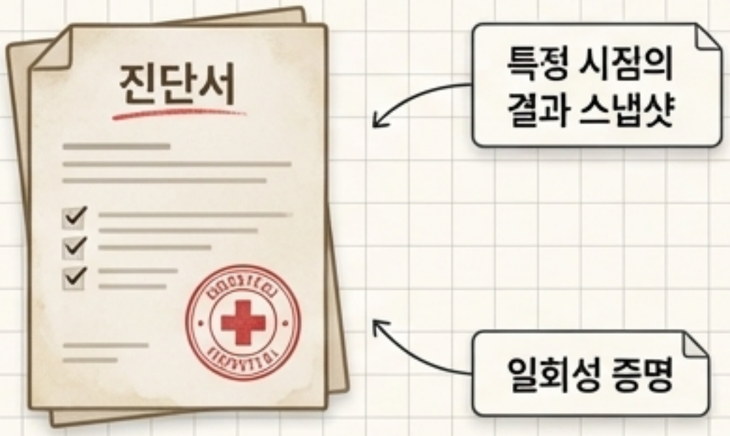
- 진단은 성공적이었고 취약점은 발견되었습니다. 하지만 보고서가 고객에게 전달되고 파일이 닫히는 순간, 그 지식은 어딘가로 사라집니다.
- 다음 진단자는 처음부터 다시 시작하며, 6개월 전의 분석 과정을 똑같이 반복해야 합니다.
- 문제는 보고서의 내용이 아니라, '정적 문서'라는 매체 그 자체입니다.

형식의 진화가 본질적 한계를 덮을 수는 없습니다

	Word/PDF (내용보다 분량)	Excel (파편화된 정량화)	포털 시스템 (경직된 중앙화)	Git 저장소 (STaC)
맥락 보존력 (Context Retention)	낮음 (일반 가이드라인 위주)	매우 낮음 (결과만 존재)	낮음 (정해진 필드에 국한)	매우 높음 (코드 주석 및 커밋 히스토리)
재현성 (Reproducibility)	불가	불가	불가	즉시 실행 가능 (PoC 코드)
버전 관리 (Version Control)	수동 (v1.1_final.pdf)	이메일/로컬 분산	DB 의존적	Git 기반 완벽한 추적
상호운용성 (Interoperability)	단절	필터/정렬 가능	API 제한적	CI/CD 및 SARIF/SBOM 완벽 통합

보안 진단은 진단서가 아니라 의무기록이 되어야 합니다

진단서 (Medical Certificate)



의무기록 (Medical Record)

지금까지의 보안 보고서는 그 순간의 결과만 담은 '진단서'였습니다. 우리에게 진짜 필요한 것은
다음 작업자가 맥락을 완벽히 이어받을 수 있는 살아있는 '의무기록'입니다. 문서는 발행되면 죽지만, 기록은 이어집니다.

문서 중심 보고에서 버전 관리 기반 검증으로의 통합

Word/PDF 시대: 모의해킹 산출물의 지배적 형태.

Excel 템플릿: 집계는 개선되었으나 시간 경과에 따라 데이터 파편화 발생.

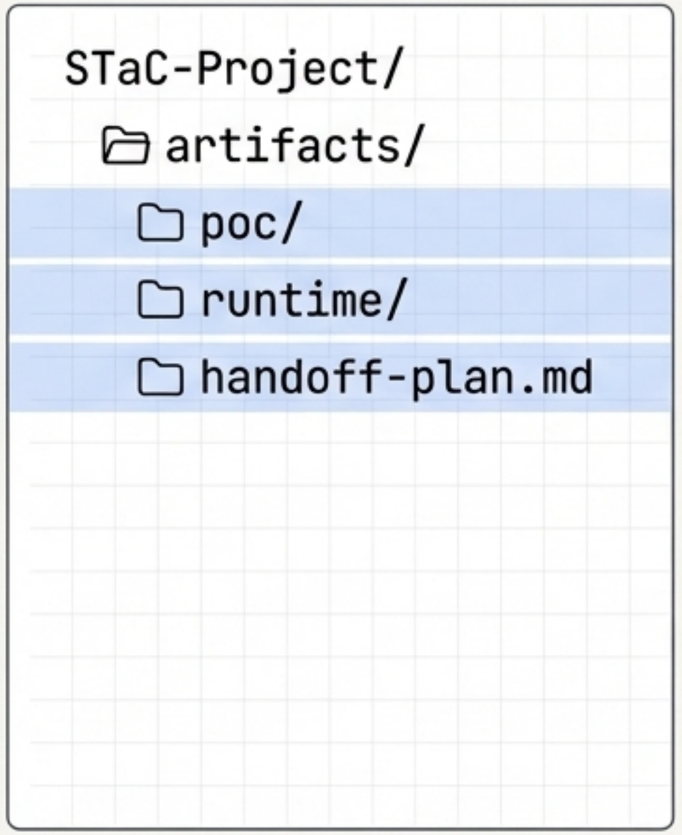
보안 포털/DB: 발견 사항의 중앙화. 그러나 새로운 유형 반영에 경직됨.

GitOps 대중화: PR 기반 운영 체계 확립. 운영의 유일한 진실의 원천(SSOT)으로 Git 부상.

CI/CD 보안 내재화: 파이프라인 내 스캔 및 게이팅 통합.

Security Testing as Code (STaC): 저장소 + 파이프라인 + 증거 번들의 실용적 종합.

주장(Claims)에서 실행 가능한 증거(Evidence)로의 전환



항목	기존 (서술형 주장)	STaC (실행 가능한 증거)
취약점 설명	A 함수에서 XSS가 발생할 수 있습니다.	artifacts/poc/ 내 구동 가능한 Fuzz Harness 작성
취약점 증명	스크린샷 이미지 캡처 첨부	artifacts/runtime/ 내 캡처된 실제 HTTP 트래픽 및 curl 로그
후속 조치	입력값을 검증하세요 (일반론)	handoff-plan.md를 통한 재현 스크립트 및 후속 검증 매트릭스 제공

**“구조 자체보다 각 폴더 안에 ‘무엇’이 들어가는지가 핵심입니다.
진단 결과는 주장이 아니라, 버전 관리가 가능한 1급 파이프라인 아티팩트(Artifact)여야 합니다.”**

사례 1: 취약점 서술을 퍼징 테스트 코드로 대체하다

The Dead Way (Word)

XSSStringSerializer 클래스가 unescape를 수행하여 설계와 달리 XSS 방어가 무력화될 수 있는 우기 위험이 존재하므로 수정이 필요합니다.

The Living Way (STaC: `artifacts/poc/`)

JazzerFuzzHarness.java

```java
import com.code_intelligence.jazzer.api.FuzzedDataProvider;

public class JazzerFuzzHarness {
    public static void fuzzerTestOneInput(FuzzedDataProvider data) {
        String input = data.consumeRemainingAsString();
        // Fuzzing target: XSSStringSerializer.unescape
        try {
            XSSStringSerializer.unescape(input);
        } catch (Exception e) {
            // Catch unexpected exceptions
        }
    }
}
```

```
$ make run-fuzz-xss
...
[INFO] Running fuzzer...
[INFO] Fuzzing finished.
Crash files found: 7
```

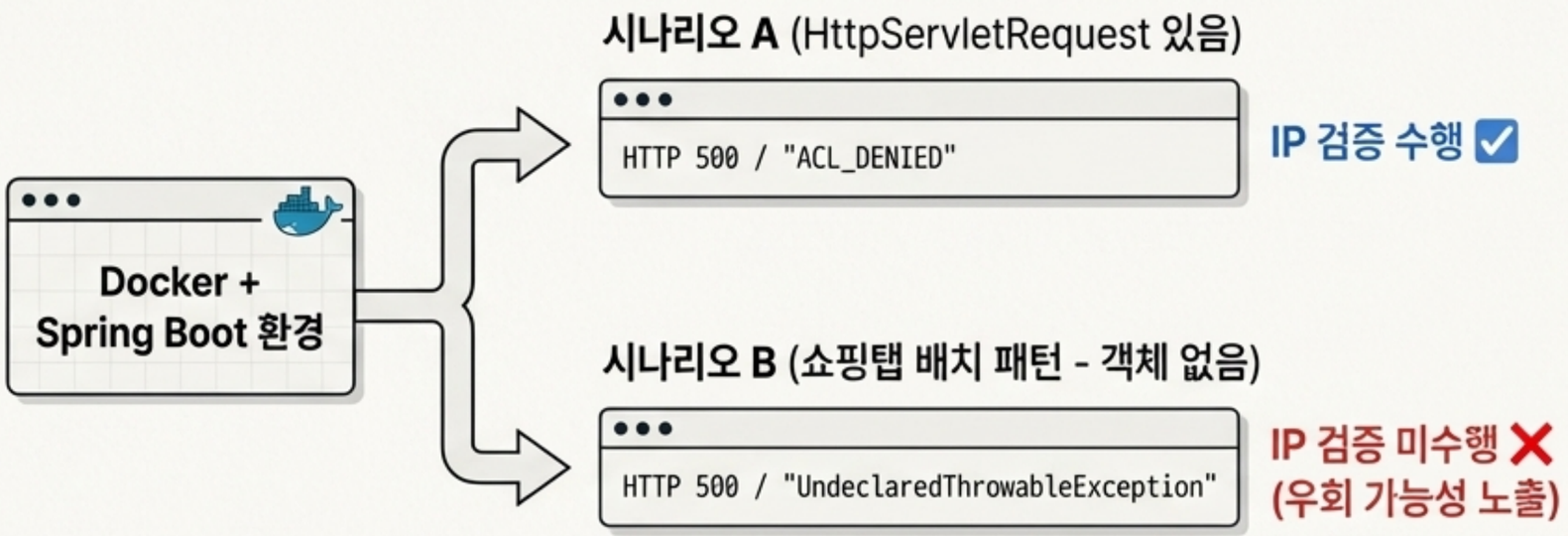
이름만 방어 로직인 XSSStringSerializer. 단 한 줄의 문장 대신 Jazzer 기반 Java 퍼저를 작성했습니다. 누구든 저장소를 clone하여 아래 명령어(**make run-fuzz-xss**) 하나만 실행하면 **26개의 페이로드 주입**과 **7개의 크래시 결과**를 동일하게 재현할 수 있습니다.

사례 2: 복잡한 로직 우회를 재현 가능한 시나리오로 박제하다

The Dead Way (Word)

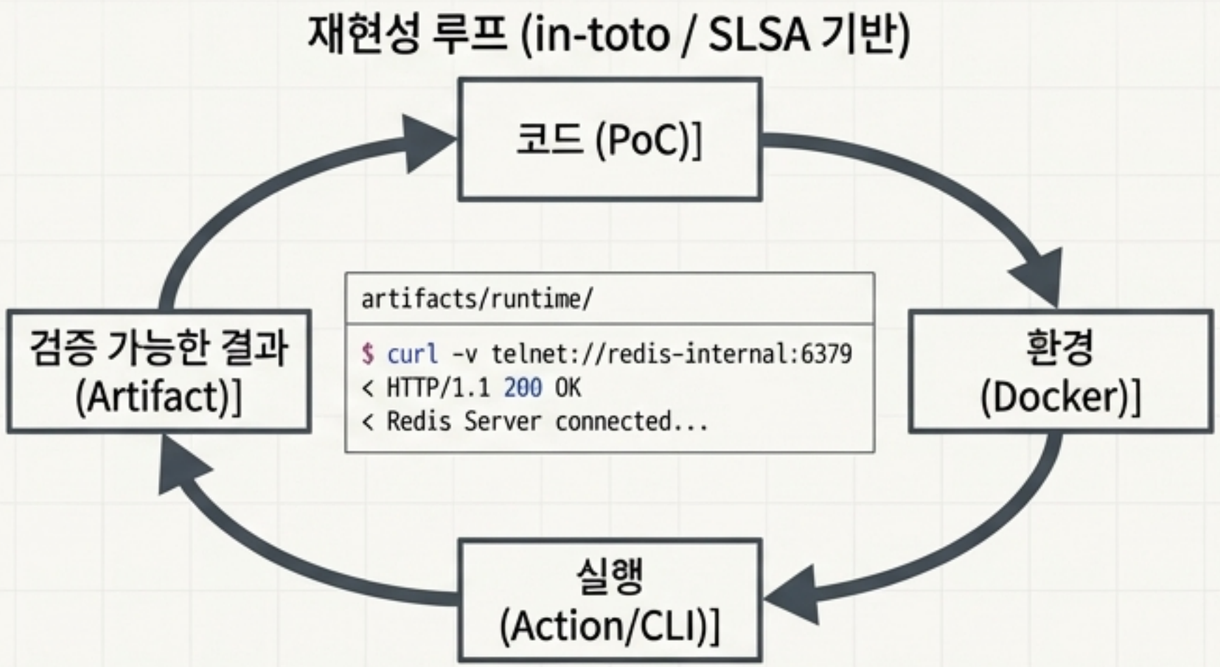
배치 컨트롤러의 설계와 구현이 불일치하여 @Acl 어노테이션의 IP 검증 로직이 우회될 가능성이 있습니다.

The Living Way (STaC: `artifacts/poc/acl-bypass/`)



비즈니스 로직은 두 경우 모두 실패하지만, 예외 처리 로직이 바뀌는 순간 보안 흠이 열립니다. 이 미묘한 차이를 "설계 불일치"라는 문장으로 퉁치는 대신, Docker 환경을 구성해 한 줄의 명령어로 재현 가능하도록 캡처했습니다.

런타임 증거: '확인했다'는 주장이 아닌 '증명'의 저장



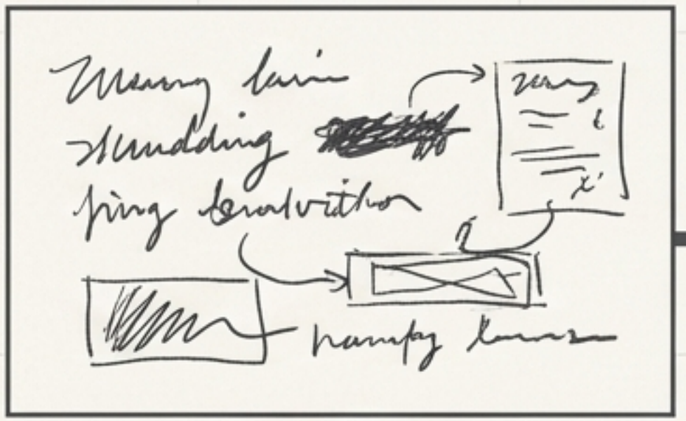
- 내부 네트워크에 상용 Redis가 TCP로 열려있음을 확인하는 과정. 비파괴 원칙을 지키기 위해 실제 인증 시도 없이 도달 가능성만 증명해야 합니다.
- 서술형 보고서는 '6노드 중 4노드 도달 확인 완료'라고 단순하게 적습니다.
- STaC 프로젝트는 `spring.redis.password` 미설정 상태를 찌르는 `curl` 명령어와 실제 반환된 Response Header를 원시 파일로 보존합니다. 확인한 '결과 자체'가 프로젝트 안에 남습니다.

완벽한 지식의 인수인계: handoff-plan.md

시나리오 (Scenario)	상태 (Status)	대상 커밋 해시 (Target Commit Hash)	후속 작업 (Next Steps)
XSS 방어 해제 검증	완료 (퍼징)	`a1b2c3d`	Phase B: 추가 페이로드 확장
배치 API ACL 우회	대기 (동적 검증)	`a1b2c3d`	Phase A: 예외 처리 로직 패치 후 재실행
Redis 인증 누락	완료 (런타임 증거)	인프라	Phase C: 내부망 방화벽 규칙 검토

다음 진단자는 결코 처음부터 시작하지 않습니다. 어떤 시나리오가 코드로 확인되었는지, 무엇이 범위 밖으로 명시적으로 제외되었는지 매트릭스로 정리됩니다.
분석 대상은 커밋 해시로 영구 고정되어 6개월 후에도 완벽한 환경 재현이 가능합니다.

AI & LLM: 서술형 진단을 살아있는 코드로 변환하는 경제적 촉매제



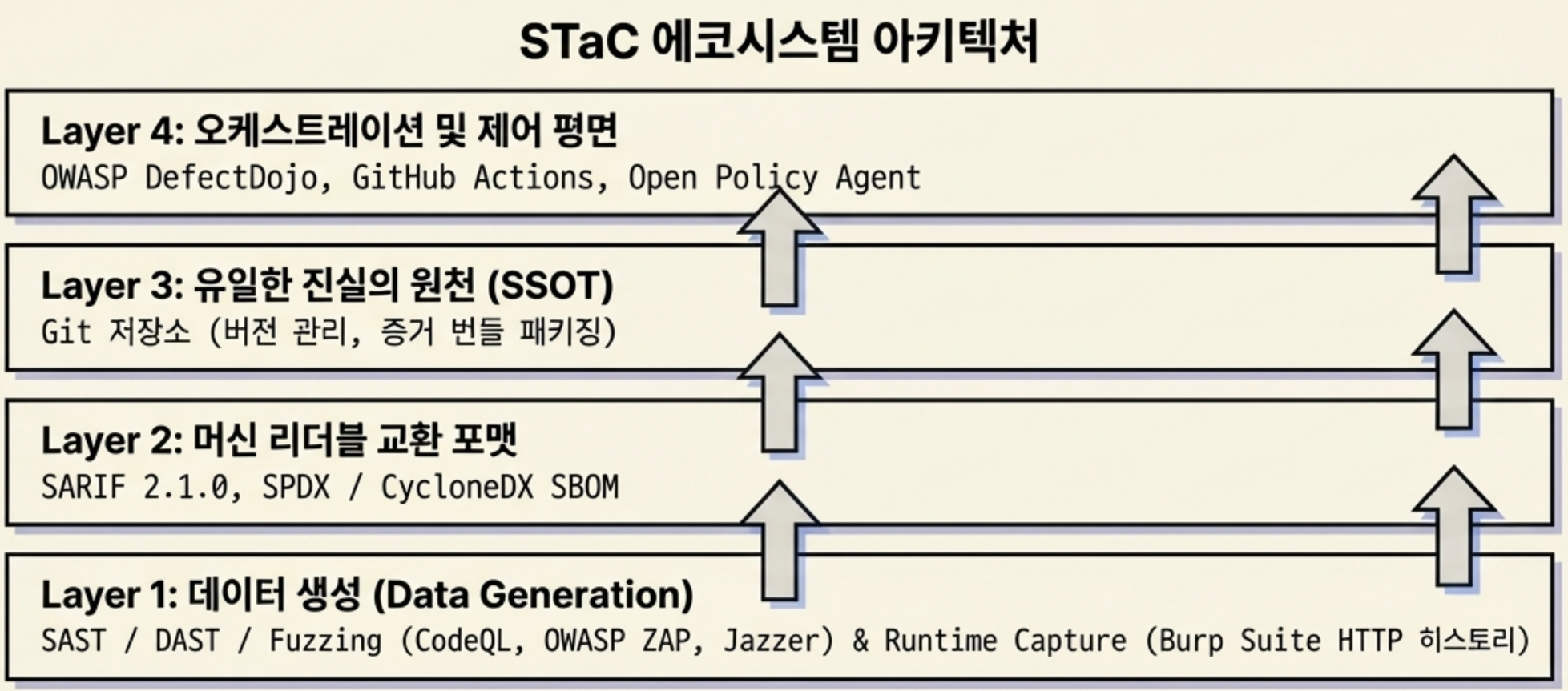
비정형 분석 결과
(진단자의 머릿속)

LLM
(TestPilot 모델 접근)

- 구조화된 문서화
- 취약점 시나리오 ➡ 테스트 케이스 변환
- PoC 코드 초안 생성

- 이 방식은 AI 이전에는 경제적으로 불가능했습니다. 수동으로 보고서를 쓰는 시간보다 코드를 짜는 시간이 더 걸렸기 때문입니다.
- AI를 '보고서 예쁘게 써주는 도구'로 쓰면 그저 약간 나은 Word 파일이 나올 뿐입니다.
- 하지만 AI를 '비정형 진단 지식을 구조화하는 도구'로 활용하면, 복잡한 취약점 시나리오를 실행 가능한 테스트 케이스로 즉시 변환할 수 있습니다. 실행 결과에 의한 철저한 검증으로 AI의 환각(Hallucination)도 원천 차단됩니다.

현대적 에코시스템: STaC를 가능하게 하는 표준과 도구들



이 개념은 새롭게 발명된 것이 아닙니다. 이미 성숙한 산업 표준과 도구 체계가 서술 산문(Prose)을 구조화된 데이터(Data)로 변환하는 계층화된 패키징 모델을 완벽하게 지원하고 있습니다.

학술 연구와 글로벌 산업계가 증명하는 필연적 미래

학술적 검증 (Academic)
in-toto (USENIX 2019)
& Reproducible Builds

소프트웨어 전달 무결성의 검증 가능한 공식화. 증거는 주장이 아니라 재구성 가능한 과정이다.

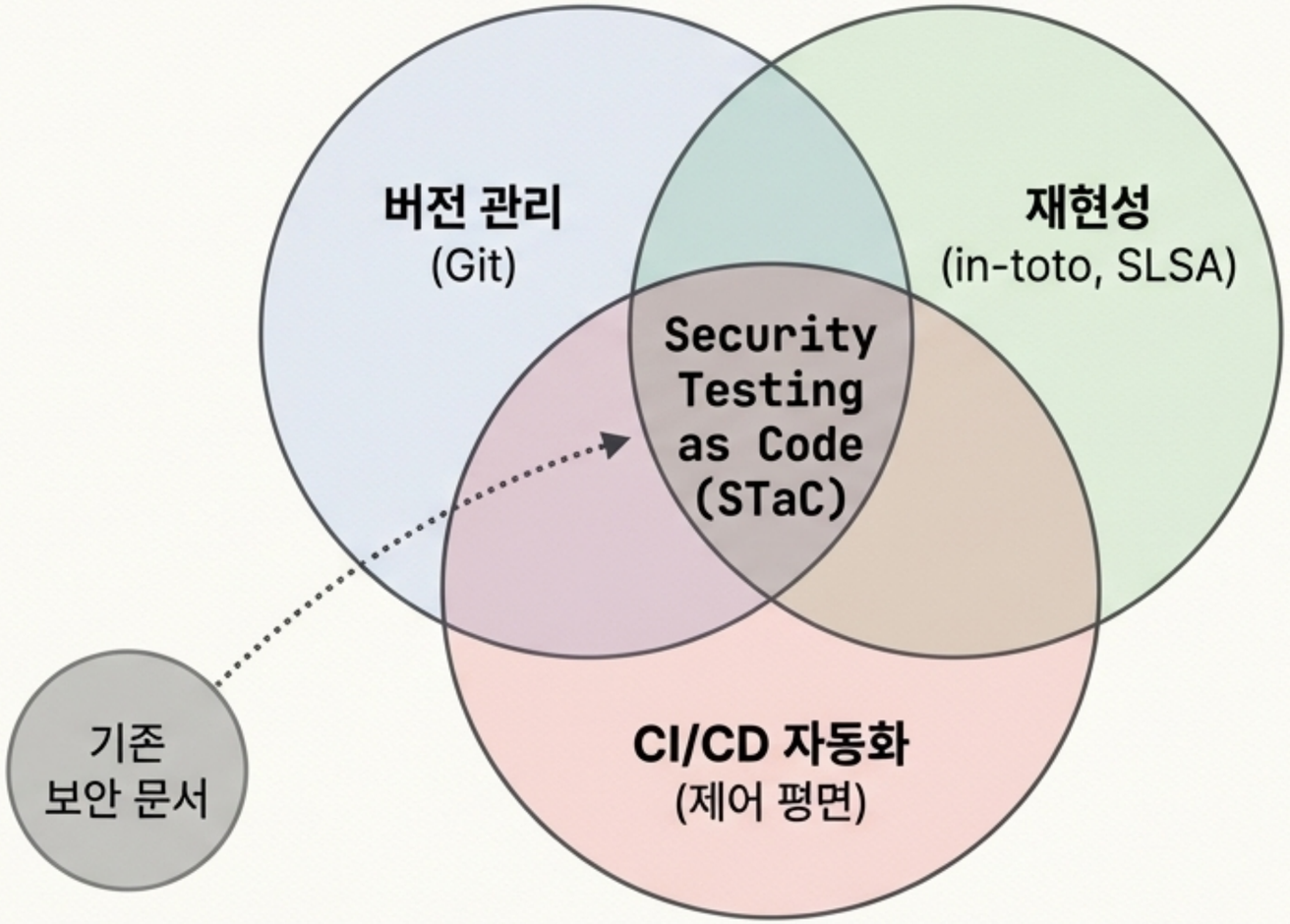
산업의 실증 (Industry)
Google OSS-Fuzz &
Elastic detection-rules

10,000개 이상의 취약점 식별. 퍼즈 타겟과 실행을 사후 첨부가 아닌 1급 파이프라인 아티팩트로 취급.

통제 평면 (Control Plane)
SANS DevSecOps Survey
& Argo CD / GitOps

CI/CD를 단순 전달 수단이 아닌, 테스트 삽입 및 End-to-End 감사 추적을 위한 제어 평면으로 정의.

보안, 마침내 소프트웨어 엔지니어링의 본류로 통합되다



Security Testing as Code는 유행어가 아닙니다. 새로운 보안 도구도 아닙니다. 이는 소프트웨어 엔지니어링 생태계를 혁신한 가장 강력한 원칙들(버전 관리, 재현성, 자동화)이 마침내 보안 진단이라는 고립된 영역의 정중앙에 자리 잡는 '최종 통합'입니다.

살아있는 코드만이 영속하는 보안 지식을 만듭니다

```
$ execute_stac_manifesto.sh
> [INFO] 진단 결과는 주장이 아니라 증거여야 합니다.
> [INFO] 증거는 재현 가능해야 합니다.
> [INFO] 재현 가능한 증거는 파일이며, 파일은 버전 관리가 됩니다.
> [SUCCESS] 문서는 읽히고 끝나지만, 코드는 실행되고 이어집니다.
```

이제 보안 진단은 서재에 꽂히는 문서가 아닙니다.
CI/CD 파이프라인 위에서 호흡하며 진화하는 살아있는 코드입니다.
STaC 시대의 도래를 맞이하십시오.