OWASP SEOUL CHAPTER NIGHT · RSAC WEEK · SAN FRANCISCO

Can We Trust the Code We Ship?

Vibe Coding 시대의 AppSec

Provally 최광준(Jun)

Speaker & Company

최광준 (Kwangjun Choi)

전 S2W 오펜시브 리서처

20개 이상의 제로데이 취약점 발견

해킹보안연구팀 Hackyboiz 설립 및 운영

PROVALLY

AI 기반 SAST 검증 자동화 플랫폼 AutoProof

→ PoC 자동 생성 및 검증

SAST finding의 실제 exploitable 여부 PoC 생성 및 검증

→ 격리 인프라 자동 구성

테스트 환경 자동 구성 및 PoC 안전 실행

→ AI 기반 SAST Rule 관리

프로젝트 맞춤형 SAST rule 생성·튜닝·관리

AI-assisted Development는 이미 기본 Workflow

55.8%

더 빠른 작업 완료

GitHub Copilot 사용 개발자 태스크 완료 속도 향상

42%

AI 생성 코드

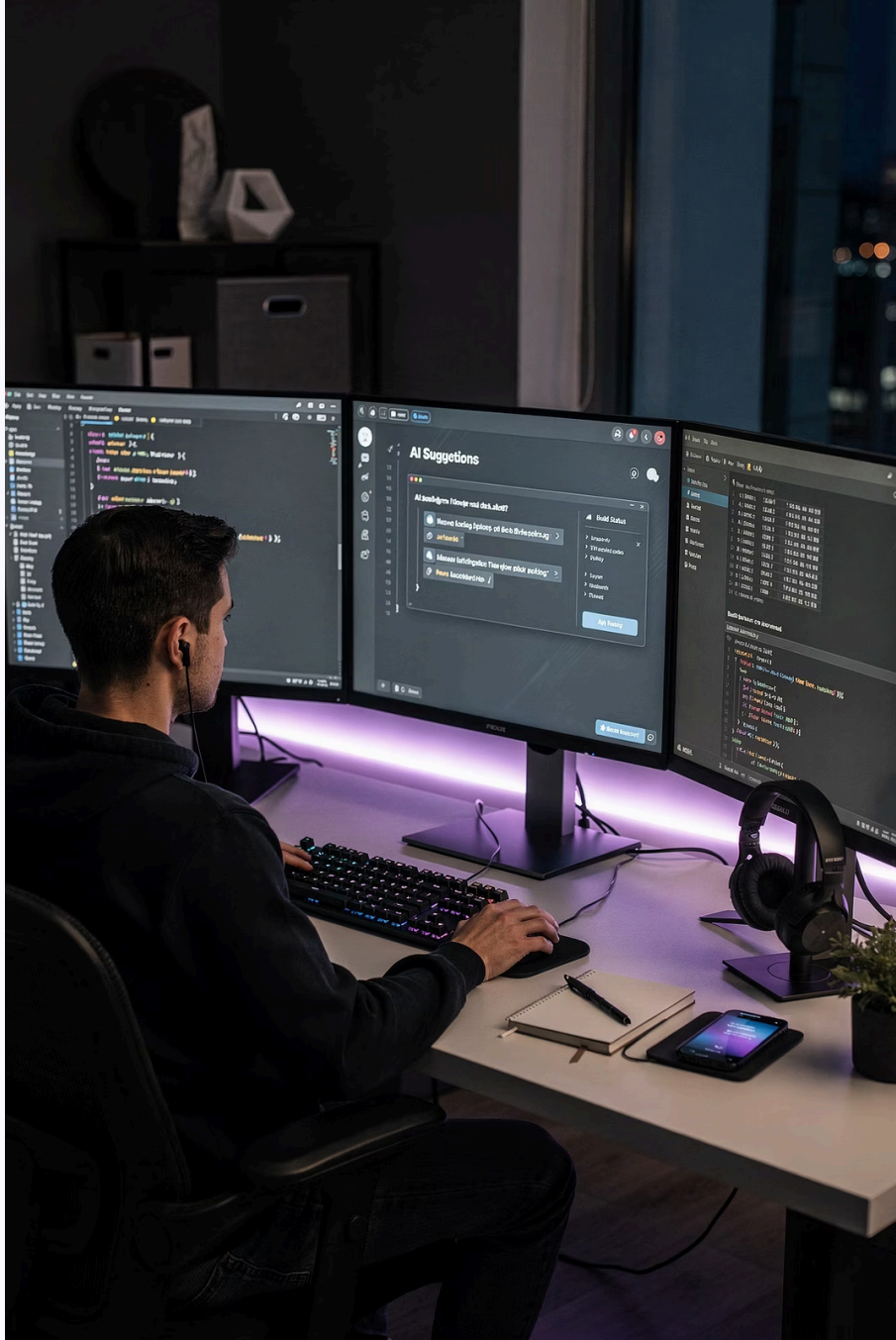
Sonar 2026 기준, committed code 중 AI-generated or AI-assisted 비율

50.6%

Daily AI 사용

Stack Overflow 2025: 전문 개발자 중 AI 툴 매일 사용 비율

이제 AppSec의 질문은 **"왜 이렇게 빨라졌나?"**에서 **"이 속도를 어떻게 검증할 것인가?"**로의 전환



속도는 올라갔지만, 신뢰는 그만큼 올라가지 않았다

Stack Overflow 2025 데이터

33%

AI output 신뢰 응답 개발자 비율

66%

"almost right, but not quite"가 가장 큰 불만이라는 응답

Stanford 연구가 보여주는 역설

AI 어시스턴트 사용 시 **더 insecure한 코드 작성**과 동시에 **더 secure하다고 믿는 경향**

문제는 코드 생성 속도가 아니라 **검증 속도**다. 오늘 이야기의 핵심은 바로 이 **Trust Gap**이다.

취약점 유형은 새롭지 않지만, AI가 증폭

Broken Auth & Authz
인증·인가 로직의 결함

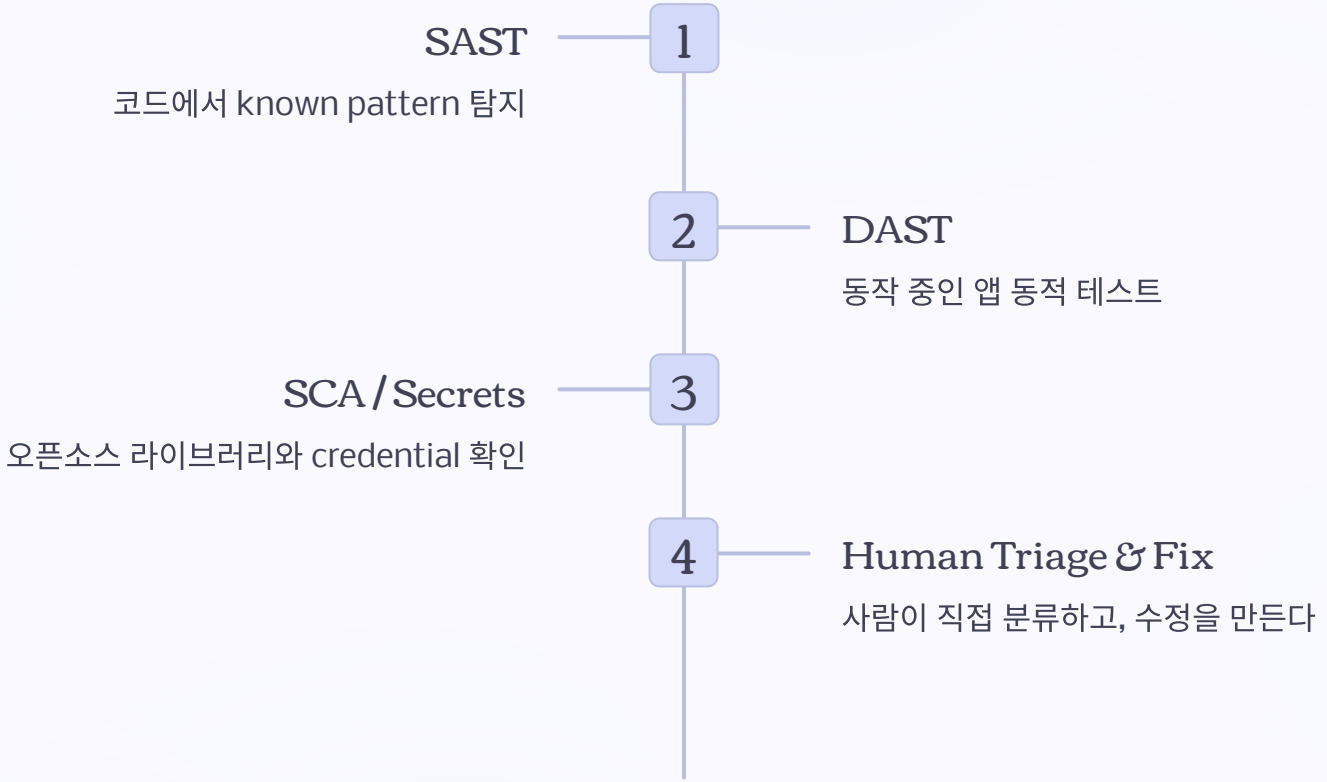
Unsafe Input & Data Handling
입력값 검증 누락 및 불안전한 데이터 처리

Secrets & Config Leakage
자격증명·설정값 노출

Dependencies & Supply Chain
오픈소스 및 공급망 리스크

GitGuardian 2026: AI-assisted 커밋에서 leaked secrets 발생률 약 **2배**. AI는 완전히 새로운 버그보다, 익숙한 취약점을 더 빠르게 더 많이 생성

이전 AppSec은 비교적 단순했다



하지만 AI coding으로 **code volume**과 **verification cost**가 동시에 증가, scanner 하나로 설명되지 않는 복잡한 시장이 형성 중

시장의 대응은 다층적으로 갈라지고 있다

핵심 질문 – 어디서 제어할 것인가

- 어디서 더 빨리 **고칠** 것인가
- 어디서 더 잘 **리뷰**할 것인가
- 어디서 더 깊게 **reasoning**할 것인가
- 어디서 애초에 위험한 코드를 덜 **만들** 것인가

- AI Autofix
- LLM Security Reviewers
- Business Logic Bug Detection
- Architecture / Design Risk Detection
- Secure-coding guardrails at generation time

카테고리 1

AI Autofix

대표 플레이어

- GitHub Copilot Autofix
- Snyk Agent Fix
- Semgrep Autofix

Finding 이후의 Fix를 빠르게

역할

기존 scanner workflow 위에 바로 얹은 remediation speed 즉각 향상. 가장 현실적인 진입 계층.

한계

alert를 닫는 것과 real risk를 줄이는 것은 다를 수 있음. subtle logic flaw나 behavior regression은 여전히 사람 검토 필요.

단순 패치 코드 생성? 검증이 필요하다!

카테고리 2

LLM Security Reviewers

대표 플레이어

- Claude Code Security
- Codex Security

Context를 읽고 Reasoning하는 리뷰어

역할

단순 rule match를 넘어서 repo context, PR, code change, threat boundary 분석. 설명 가능한 finding과 suggested patch 제공.

의미

scanner-first review에서 **context-aware review**로 이동하는 전환점. 단, human review와 최종 validation 대체 불가.

카테고리 3

비즈니스 로직 취약점 탐지

대표 플레이어

- AppSentinels
- ZeroPath
- Semgrep Multimodal

악용 가능한 흐름 직접 탐지

취약한 코드 조각보다 **악용 가능한 흐름 전체** 분석. API, agent, MCP, runtime action까지 함께 검토.

Auth Bypass / BOLA / BFLA

인증·인가 우회 및 객체 수준 접근 제어 결함

IDOR / Workflow Abuse

직접 객체 참조 및 비즈니스 로직 남용

Payment Flow Race

결제·트랜잭션 경쟁 조건 취약점

pattern matching 중심 AppSec에서 **exploitability와 abuse path 중심 AppSec**으로 이동하는 핵심 계층

카테고리 4

Architecture / Design Risk Detection

대표 플레이어

- **Clover**
- **Apiiro**

이 계층의 역할

- 코드가 나오기 전에 **design risk** 선제적 분석
- design-to-code drift 탐지
- requirement, policy, architecture, runtime exposure 통합적 관점

"취약한 코드가 있나?" 를 묻는 것에서

"우리가 위험한 설계를 하고 있나?"

를 먼저 묻는 방향으로의 근본적 전환이다.

카테고리 5

Secure-coding Guardrails

생성 시점의 예방 제어

대표 플레이어

- Corridor
- OpenSSF Guidance
- AI Instruction / Rulefile / IDE Governance 도구들

AI가 처음부터 덜 위험한 코드를 쓰게 만드는 제어

→ Generation 시점 Context 주입

coding agent에게 security context와 rule 직접 제공, 생성 결과물 제어

→ 사전 Prevention 시도

생성 시점부터 불안전한 패턴의 코드 진입 차단

→ AppSec 영역 확장

사후 탐지를 넘어 **generation control**까지 AppSec 책임 범위 확장

그래서 앞으로의 AppSec은 어떻게 달라지는가

🔧 Autofix

Remediation Speed

발견된 취약점 신속 수정

🔍 Reviewer

Context-aware Triage

문맥 이해 지능형 분류

⚡ Logic

Abuse Path Detection

실제 악용 경로 중심 탐지

🏗️ Architecture

Design Risk Prevention

코드 이전 단계 설계 리스크

🛡️ Guardrails

Unsafe Generation Reduction

생성 시점 예방 제어

AI 시대의 AppSec은 **scan + review + reason + govern**이 함께 돌아가는 다층 구조가 된다.

마무리

예전의 질문

> "무슨 취약점이 있나?"

지금의 질문

> "어느 계층에서 취약점을 줄이고, 어디서 검증할 것인가?"

LLM은 AppSec을 없애지 않음

오히려 AppSec을 더 다층적이고 운영형인 분야로 변화

더 많은 취약점, 더 이른 시점

더 다양한 관점에서, 더 이른 시점에 줄이는 방향으로 진화

Thank you 🙏

https://provally.io