ARAŞTIRMA RAPORU · 2026

Vibe Coding Güvenlik Raporu

"Açıkları kapat" demek neden yetmiyor?

10 projede iki yaklaşımın karşılaştırması: işi bitirip sonradan taratmak ve en baştan security md ile geliştirmek. Sektördeki tablo, saldırgan bakışı ve uygulanabilir öneriler.



Sedat Özdemir

Sr. Expert Pentest & Vulnerability Management Engineer

Kasım 2026 · v1.0

sedatozdemir.com.tr

RAPORUN AKIŞI

İçindekiler

Önsöz 3
Bu raporun nereden çıktığı ve kimin için yazıldığı

Yönetici özeti 4
Ana bulgular ve rakamlar tek sayfada

01 Sektörde durum: herkes vibe coding yapıyor 5
Vibe coding'in iki yılda nasıl varsayılan yol haline geldiği

02 Zafiyetler neden artıyor? 8
Modelin, prompt'un ve hızın hatayı nasıl büyüttüğü

03 Yapay zekâ geliştikçe saldırı yüzeyi neden büyüyor? 11
Agent'lar, kural dosyaları, hallucinated paketler ve hızlanan saldırgan

04 Araştırma: kapsam ve yöntem 14
10 proje, 4 durum, 3 test yöntemi

05 Bulgular 17
Genel tablo, hata sınıfları, kategori ısı haritası, proje bazında sonuçlar

06 Tara-kapat neden yetmiyor? 22
Dört neden, iki vaka ve prompt'u tekrar etmenin sınırı

07 Security md ne değiştiriyor? 25
Kullandığım dosya, üç gözlem ve kısa dosyanın neden daha iyi çalıştığı

08 Claude bu güvenliği nasıl sağlıyor? 28
Claude Code'daki katmanlar ve sınırları

09 Saldırgan gözüyle: güvenlikçi bakış açısı 31
Bir vibe coding uygulamasını test ederken nereden başladığım

10 Sonuç 33
Hikâyenin özeti ve önümüzdeki dönem

11 Öneriler 35
Geliştiriciler ve ekipler için uygulanabilir adımlar

Ekler 37
Sınırlılıklar, sözlük, kaynaklar ve hazır md'ler

NEDEN BU ÇALIŞMAYI YAPTIM?

Önsöz

Son bir yılda test ettiğim uygulamaların giderek daha büyük bir kısmı, ekiplerin "AI ile birkaç haftada çıkardık" dediği projelerden oluşuyor. Açılış toplantısında da genelde aynı cümleyi duyuyorum: "Yayına almadan önce Claude'a tarattık, bulduğu açıkları kapattı." Testin ilk bir saatinde ise çoğu zaman aynı tabloyla karşılaşıyorum. Kod temiz görünüyor, linter mutlu, test'ler geçiyor; ben de URL'deki ID'yi değiştirerek başka bir kullanıcının faturasını açabiliyorum.

Bu çelişki bir süredir aklımı kurcalıyordu. Tarama gerçekten işe yarıyordu, çünkü injection ve hardcoded secret gibi bulgular belirgin şekilde azalmıştı. Yine de bir saldırgan için en değerli olan hatalar yerinde duruyordu. Merak ettiğim soru basitti: "Projeyi tara ve açıkları kapat" demek sorunun ne kadarını çözüyor, ve kuralları en baştan yazmak gerçekten fark yaratıyor mu?

Bunu tahminle değil ölçerek cevaplamak istedim. 10 farklı stack'te aynı uygulamaları dört farklı yoldan ürettim ve her birini aynı testten geçirdim. Sonra da sonuçları sektörde olup bitenle yan yana koydum, çünkü benim lab'ımda gördüğüm şey tek başına bir anekdot olarak kalabilirdi. Kalmadı. Veracode'dan Apiiro'ya, Escape'ten CrowdStrike'a kadar farklı kaynaklar aynı yöne işaret ediyor.

Raporu üç gruba yazdım. Vibe coding ile ürün çıkaran geliştiriciler, neyi neden yapmaları gerektiğini görsün istedim. Bu ekiplerle çalışan AppSec ve pentest ekipleri, "AI taradı" cümlesinin arkasında neyin kaldığını bilsin. Yöneticiler de hızın bedelinin nerede ödendiğini rakamlarla görebilsin.

Deneyin örneklemi küçük ve bunu Ekler bölümünde açıkça yazdım. Yine de çıkan yön oldukça net: zafiyeti kapat demek yeterli değil. Kuralı en baştan, modelin her kod yazışında göreceği yere yazmak gerekiyor.



Sedat Özdemir
Sr. Expert Pentest & Vulnerability Management Engineer
sedatozdemir.com.tr

BİR BAKIŞTA

Yönetici özeti

Vibe coding, kod üretme hızını birkaç kat artırdı. Güvenlik tarafı aynı hızla büyümedi ve ekiplerin büyük kısmı bu açığı işin sonunda tek bir prompt ile kapatmaya çalışıyor: "Projeyi tara ve açıkları kapat." Bu rapor o prompt'un ne kadar işe yaradığını 10 projede ölçüyor ve en baştan yazılan bir security md ile karşılaştırıyor.

DENEYİN ÖNE ÇIKAN RAKAMLARI

214
Ham halde 10 projede bulunan toplam bulgu; 61'i kritik ya da yüksek

%55
Tara-kapat prompt'unun kapattığı bulgu oranı

%73
Tara-kapat sonrasında yerinde kalan tasarım hatası oranı (authorization, IDOR, iş mantığı)

13
Fix sırasında modelin ürettiği yeni bulgu

%81
Security md ile baştan geliştirilen sürümlerde ham hale göre daha az bulgu

11 → 58 dk
Saldırgan gözüyle ilk kritik/yüksek bulguya ulaşma süresi, ham ve security md

Dört ana bulgu

1. **Tara-kapat işin yarısını yapıyor.** Bulguların %55'i kapandı. Kapananların neredeyse tamamı, koda bakınca görünen kalıp hatalarıydı (injection, XSS, secret, header).
2. **Tasarım hataları yerinde kalıyor.** Eksik authorization, IDOR, tenant izolasyonu ve iş mantığı gibi bulguların %73'ü tara-kapat'tan sonra da oradaydı. Prompt'u üç kez tekrarlamak tabloyu değiştirmedi.
3. **Kuralı baştan yazmak hatanın oluşmasını engelliyor.** Security md ile geliştirilen sürümlerde tasarım hataları 96'dan 29'a indi. Review katmanı eklenince toplam bulgu 17'ye düştü.
4. **Kalan risk, işi bilen kişide.** En son kalan 17 bulgunun 10'u iş mantığıyla ilgiliydi. Bunları ne kural dosyası ne de AI review yakaladı.

Ne yapmalı?

İlk prompt'tan önce repoya kısa ve somut bir `SECURITY.md` koyun, `CLAUDE.md` içinden referans verin ve her özellikten sonra bir review katmanı çalıştırın. Mevcut projelerde tara-kapat ile başlamak mantıklı, fakat bulguları kurala çevirip md'ye yazmadan iş bitmiş sayılmamalı. İş mantığını ise kimseye devretmeyin. Ayrıntılı öneriler 11. bölümde.

01

Sektörde durum: herkes vibe coding yapıyor

Bir yıl önce şaka gibi duyulan bir terim, bugün yazılım geliştirmenin varsayılan yollarından biri. Kod üretimi hızlandı; review ve güvenlik aynı hızla büyümedi.

%84 geliştiricinin AI araçlarını kullandığı ya da kullanmayı planladığı oran (Stack Overflow Developer Survey 2025)

Bir tweet'ten varsayılan yola

"Vibe coding" terimini Andrej Karpathy Şubat 2025'te ortaya attı: koda pek bakmadan, ne istediğini anlatıp modelin ürettiğini kabul ederek ilerlemek. Terim o kadar hızlı yayıldı ki aynı yılın sonunda Collins Dictionary onu yılın kelimesi seçti. Asıl değişim ise kelimede değil, rakamlarda görünüyor.

SEKTÖR BİR BAKIŞTA

%84
geliştirici AI araçlarını kullanıyor ya da kullanmayı planlıyor
STACK OVERFLOW 2025

%46
geliştirici AI çıktısının doğruluğuna güvenmiyor; bir yıl önce %31'di
STACK OVERFLOW 2025

%25
YC W25 girişimlerinin kod tabanının %95'i AI tarafından yazılmış
Y COMBINATOR, MART 2025

%45
test görevinde AI'ın ürettiği kod güvenlik kontrolünü geçemedi
VERACODE 2025

10x
AI kod asistanı kullanan ekiplerde aylık güvenlik bulgusu artışı (6 ayda)
APIIRO 2025

2.000+
5.600 vibe coding uygulamasında bulunan zafiyet; 400'ün üzerinde açık secret
ESCAPE, EKİM 2025

Bu tablodaki en dikkat çekici çelişki ilk iki satırda: geliştiricilerin neredeyse tamamı AI kullanıyor, fakat yarısına yakını çıktıya güvenmiyor. Güvenmediğiniz bir kodu yine de yayına alıyorsanız, aradaki boşluğu bir şeyin doldurması gerekiyor. Çoğu ekipte o şey, işin sonunda yazılan "tara ve kapat" prompt'u oluyor.

Kim kod yazıyor?

Değişen sadece hız değil, kod yazan kişinin profili de. Lovable, Bolt ve Replit gibi platformlar sayesinde daha önce hiç backend yazmamış biri birkaç saatte Supabase ya da Firebase üzerinde çalışan bir uygulama yayına alabiliyor. Bu harika bir şey; fikirden ürüne giden yol hiç bu kadar kısa olmamıştı. Fakat bu kişi, row level security'nin ne olduğunu ya da bir endpoint'in neden sahiplik kontrolü yapması gerektiğini bilmiyorsa, model de ona bunu kendiliğinden söylemiyor.

ŞUBAT 2025
Karpathy "vibe coding" terimini kullanıyor
Koda bakmadan, modelin ürettiğini kabul ederek geliştirme.

MART 2025
Y Combinator W25 dönemi
Girişimlerin %25'inde kod tabanının %95'i AI tarafından yazılmış.

MAYIS 2025

Lovable uygulamalarında CVE-2025-48757

Taranan 1.645 uygulamanın yaklaşık 170'inde veritabanı tabloları public anon key ile okunabiliyor.

TEMMUZ 2025

Veracode GenAI Code Security Report

100'den fazla model, 80 görev; güvenli ve güvensiz seçenek arasında model %45 oranında güvensiz olanı seçiyor.

EYLÜL 2025

Apiiro araştırması

AI kullanan geliştiriciler 3–4 kat fazla commit atıyor; privilege escalation yolları %322 artıyor.

EKİM 2025

Escape taraması

5.600 vibe coding uygulamasında 2.000'den fazla zafiyet ve 400'ün üzerinde açık secret.

KASIM 2025

Anthropic tehdit raporu

Devlet destekli bir grup, saldırı kampanyasının taktik işlerinin büyük kısmını Claude Code'a yaptırmaya çalışıyor.

2026

CrowdStrike Global Threat Report

AI kullanan saldırganların saldırıları bir yılda %89 artıyor.

VAKA

Lovable ve CVE-2025-48757

Mayıs 2025'te bir araştırmacı Lovable'ın vitrinindeki 1.645 uygulamayı taradı ve yaklaşık 170'inde Supabase tablolarının uygulamanın içinde zaten açık duran public anon key ile okunabildiğini, bazılarında yazılabildiğini gösterdi. Açığa çıkan veriler arasında adresler, borç tutarları ve API key'leri vardı. Sorunun teknik adı eksik row level security. Pratikte anlamı şu: uygulamayı üreten kişi "bu veriyi kim görebilir?" sorusunu hiç sormamış, model de sormamıştı.

Bu vaka bu raporun ana fikrini küçük bir örnekte gösteriyor. Hata bir satırda yanlış yazılmış bir şey değildi; olması gereken bir kural hiç yazılmamıştı. İleride göreceğiniz gibi, tara-kapat yaklaşımının en zorlandığı hata türü tam olarak bu.

ÖNGÖRÜ

Vibe coding geri dönmeyecek ve dönmesi de gerekmiyor. Asıl soru, hızın getirdiği riski kimin ve hangi aşamada yöneteceği. Önümüzdeki dönemde güvenliğin, işin sonundaki bir kontrol olmaktan çıkıp modelin her prompt'ta okuduğu kurallara taşınacağını düşünüyorum.



02

Zafiyetler neden artıyor?

Model kötü niyetli değil. Kendisine anlatılanı yapıyor ve en olası kodu yazıyor; en olası kod da çoğu zaman en güvenli kod olmuyor.

%45 test görevinde AI'ın ürettiği kodun güvenlik kontrolünü geçemediği oran (Veracode 2025 GenAI Code Security Report)

Vibe coding projelerinde zafiyetlerin neden arttığını tek bir nedene bağlamak kolay olurdu, fakat gördüğüm tablo birkaç etkenin üst üste binmesinden oluşuyor. Her biri tek başına küçük; birlikte ise işin sonunda yapılan bir taramanın kapatamayacağı kadar büyük bir boşluk açıyorlar.

Model en olası kodu yazıyor

Bir dil modeli, eğitildiği kodda en sık gördüğü kalıbı üretme eğiliminde. İnternetteki örnek kodların büyük kısmı da güvenliği düşünerek yazılmamış: tutorial'lar, Stack Overflow cevapları, hızlı demo'lar. Veracode'un 2025 çalışması bunu ölçülebilir hale getirdi: modellere güvenli ve güvensiz bir yol arasında seçim yaptıracak 80 görev verildi ve modeller vakaların %45'inde güvensiz olanı seçti. Java'da bu oran %70'in üzerine çıktı. Daha çarpıcı olanı, yeni ve büyük modellerin bu konuda belirgin şekilde daha iyi olmamasıydı.

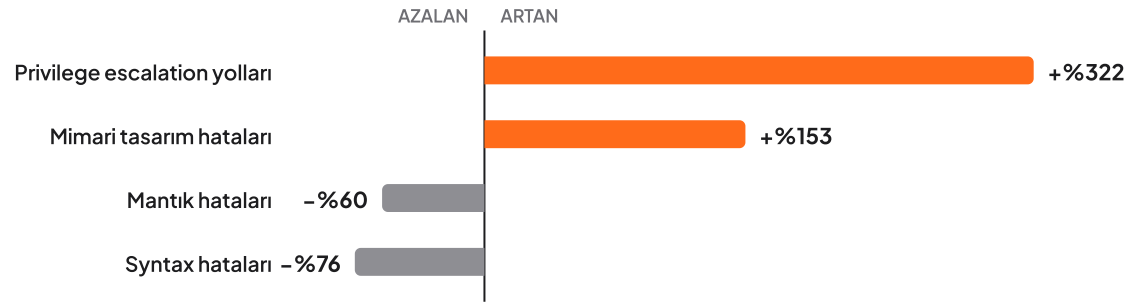
Prompt'ta olmayan gereksinim kodda da yok

Vibe coding'de gereksinimler prompt'tan geliyor. "Kullanıcılar faturalarını görebilsin" diyorsunuz ve model faturaları listeleyen bir endpoint yazıyor. "Kullanıcılar sadece kendi faturalarını görebilsin" demediyseniz, bu kısıt çoğu zaman koda girmiyor. Güvenlik, fonksiyonel olmayan bir gereksinim; kimse istemediği sürece "çalışıyor" ile "bitti" aynı şey haline geliyor. Benim deneyimde ham projelerdeki tasarım hatalarının neredeyse tamamı bu türdendi: hatalı yazılmış bir satır yoktu, istenmemiş bir kontrol vardı.

Hız, review'u geçti

Apiiro'nun Eylül 2025 araştırması, AI kod asistanı kullanan geliştiricilerin 3–4 kat fazla commit attığını, aynı dönemde aylık güvenlik bulgusu sayısının ise altı ayda on katına çıktığını gösterdi. Daha ilginç olanı, hata türlerinin nasıl değiştiğiydi.

Şekil 1. AI asistanı kullanan ekiplerde hata türlerinin değişimi



Model yazım hatasını azaltıyor, tasarım hatasını artırıyor. Bu raporun deneyinde gördüğüm ayrımın sektör ölçeğindeki karşılığı.

Kaynak: Apiiro, "4× Velocity, 10× Vulnerabilities", Eylül 2025

Bu grafik, raporun geri kalanını anlamak için önemli. Model yazım hatalarını ve basit mantık hatalarını azaltıyor; kod daha temiz görünüyor. Aynı anda privilege escalation yolları ve mimari tasarım hataları artıyor. Yani hata ortadan kalkmıyor, koda bakarak görülmesi daha zor bir yere taşınıyor. Bir kod review'cusu da, bir tarama

prompt'u da önündeki satırlara bakıyor; yazılmamış bir kontrolü görmek için sistemin nasıl çalışması gerektiğini bilmek gerekiyor.

Güven ve sahiplik bulanıklaşıyor

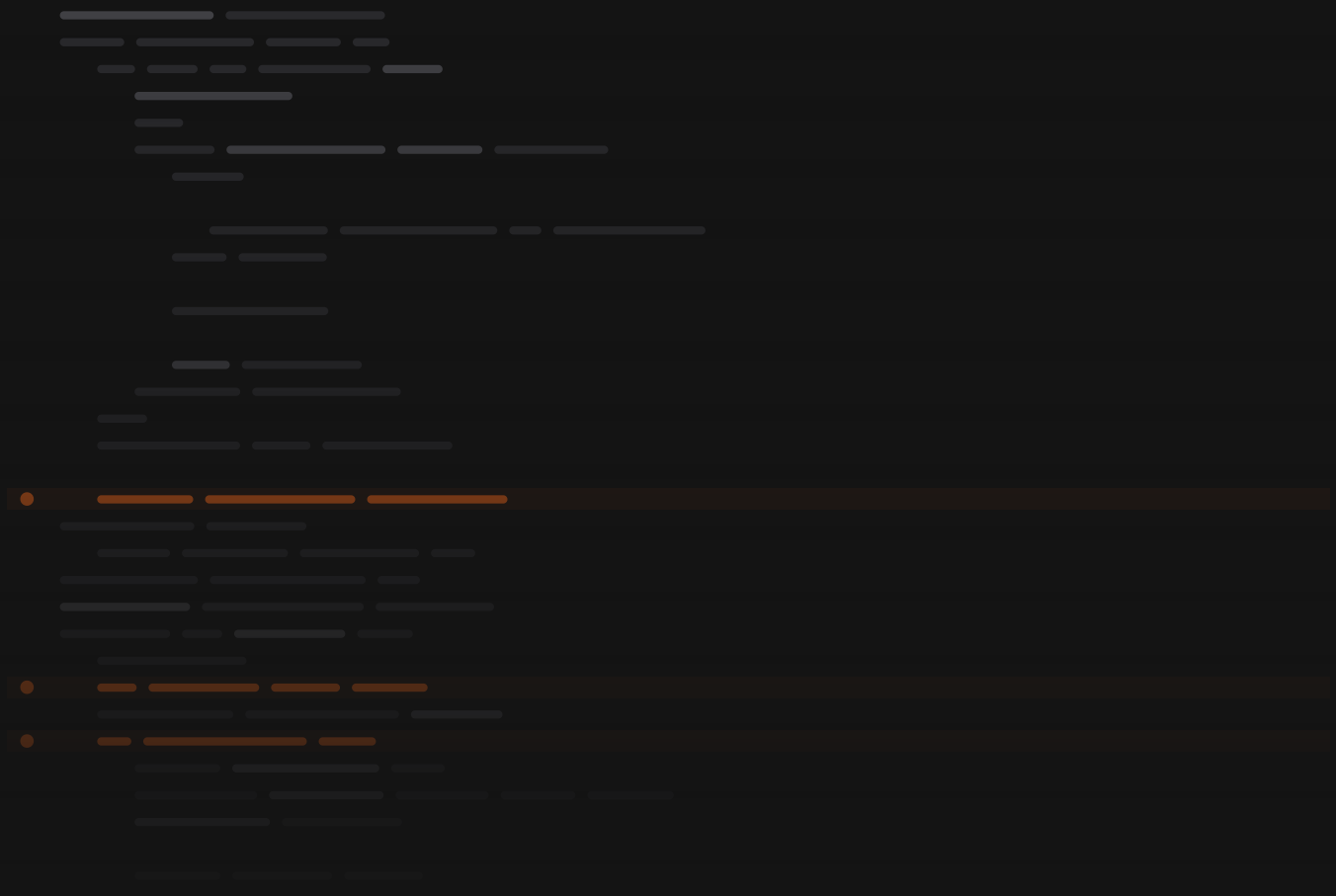
Stack Overflow'un 2025 anketinde geliştiricilerin %46'sı AI çıktısının doğruluğuna güvenmediğini söylüyor. Buna rağmen kullanım artıyor, çünkü hız kazancı gerçek. Bu durumda pratikte şu oluyor: kodu model yazıyor, geliştirici çalıştığını görüp kabul ediyor, review'cu da yüzlerce satırlık bir diff'e göz gezdiriyor. Kodun her satırını gerçekten okuyan ve "bu veri kimin?" diye soran kimse kalmıyor. Vibe coding'in tanımında zaten koda bakmamak olduğu için, bu bir disiplin eksikliği sayılmaz; yöntemin doğal sonucu.

%73

Benim deneyimde tara-kapat sonrasında yerinde kalan tasarım hatası oranı. Apiiro'nun sektör ölçeğinde gördüğü kayma ile aynı yön: model yazım hatasını temizliyor, tasarım hatası kalıyor.

ÖNGÖRÜ

Modeller daha iyi kod yazdıkça kalıp hataları azalmaya devam edecek. Tasarım hataları ise ancak kurallar yazılı hale geldiğinde azalacak, çünkü model neyin doğru olduğunu ona söylenmeden bilemiyor.



03

Yapay zekâ geliştikçe saldırı yüzeyi neden büyüyor?

Her yeni yetenek yeni bir giriş noktası getiriyor. Agent'a ne kadar yetki verirsek, saldırgan da o kadar yetkiyi hedefliyor.

%89 AI kullanan saldırganların saldırılarındaki yıllık artış (CrowdStrike 2026 Global Threat Report)

İki yıl önce bir kod asistanı, editörün içinde satır tamamlayan bir araçtı. Bugün dosya okuyan, komut çalıştıran, paket kuran, web'de arama yapan ve dış servislere bağlanan bir agent. Bu, saldırı yüzeyine yeni bir katman ekledi: kodun yanında artık kodu yazan aktörün kendisi de hedef.

Şekil 2. AI destekli geliştirmeyle eklenen saldırı yüzeyi



Model artık bir aktör

Agent bir README'yi, bir issue'yu ya da bir web sayfasını okuduğunda, o metnin içine gizlenmiş talimatlar da context'e giriyor. Prompt injection dediğimiz şey tam olarak bu: saldırgan modelle doğrudan konuşmuyor, modelin okuyacağı bir yere not bırakıyor. Agent'ın komut çalıştırma ve dosya yazma yetkisi varsa, bu not gerçek bir aksiyona dönüşebiliyor.

Kural dosyaları da bir hedef

Bu raporun önerdiği yöntem, yani `SECURITY.md` ve `CLAUDE.md` gibi kural dosyaları, kendi başına bir saldırı yüzeyi. Bu dosyalar her session'da otomatik yükleniyor ve güvenilir kabul ediliyor. Cloud Security Alliance'ın Ekim 2026 notu ve OWASP'ın AI ile güvenli kod rehberi aynı uyarıyı yapıyor: bir pull request ile kural dosyasına eklenen tek bir satır, modelin bundan sonra yazacağı bütün kodu etkileyebilir. Bu yüzden bu dosyaları CI config'i kadar ciddiye almak gerekiyor.

Hallucinated paketler ve slopsquatting

USENIX Security 2025'te yayımlanan bir çalışma, 16 modelin ürettiği 576.000 kod örneğinde önerilen paketlerin %19,7'sinin gerçekte var olmadığını gösterdi. Toplamda 205.474 farklı uydurma paket adı çıktı. Asıl risk ise tekrarlanabilirlikte: aynı prompt on kez çalıştırıldığında uydurma adların %43'ü her seferinde yeniden geldi. Bu da saldırgana bir fırsat veriyor. Modelin sık uydurduğu adları önceden npm ya da PyPI'a kaydeden biri, o paketi kuran her projeye kod çalıştırabiliyor. Buna slopsquatting deniyor.

Saldırgan da hızlandı

Saldırı yüzeyinin büyümesinin bir de öbür tarafı var. CrowdStrike'ın 2026 raporuna göre AI kullanan saldırganların saldırıları bir yılda %89 arttı. Anthropic Kasım 2025'te, devlet destekli olarak değerlendirdiği bir grubun yaklaşık 30 hedefe yönelik bir kampanyada keşif, exploit yazma ve veri toplama gibi taktik işlerin büyük kısmını Claude Code'a yaptırmaya çalıştığını ve bunu tespit edip durdurduğunu açıkladı.

Bunun vibe coding ile bağlantısı şu: aynı model aynı hatayı binlerce uygulamada tekrar ettiğinde, saldırgan için bu artık tek tek bulunması gereken zafiyetler olmaktan çıkıyor ve taranabilir bir kalıba dönüşüyor. Escape'in 5.600 uygulamalık taraması da tam olarak bunu yaptı. Bir araştırma ekibinin yapabildiğini, AI destekli bir saldırgan da yapabiliyor.

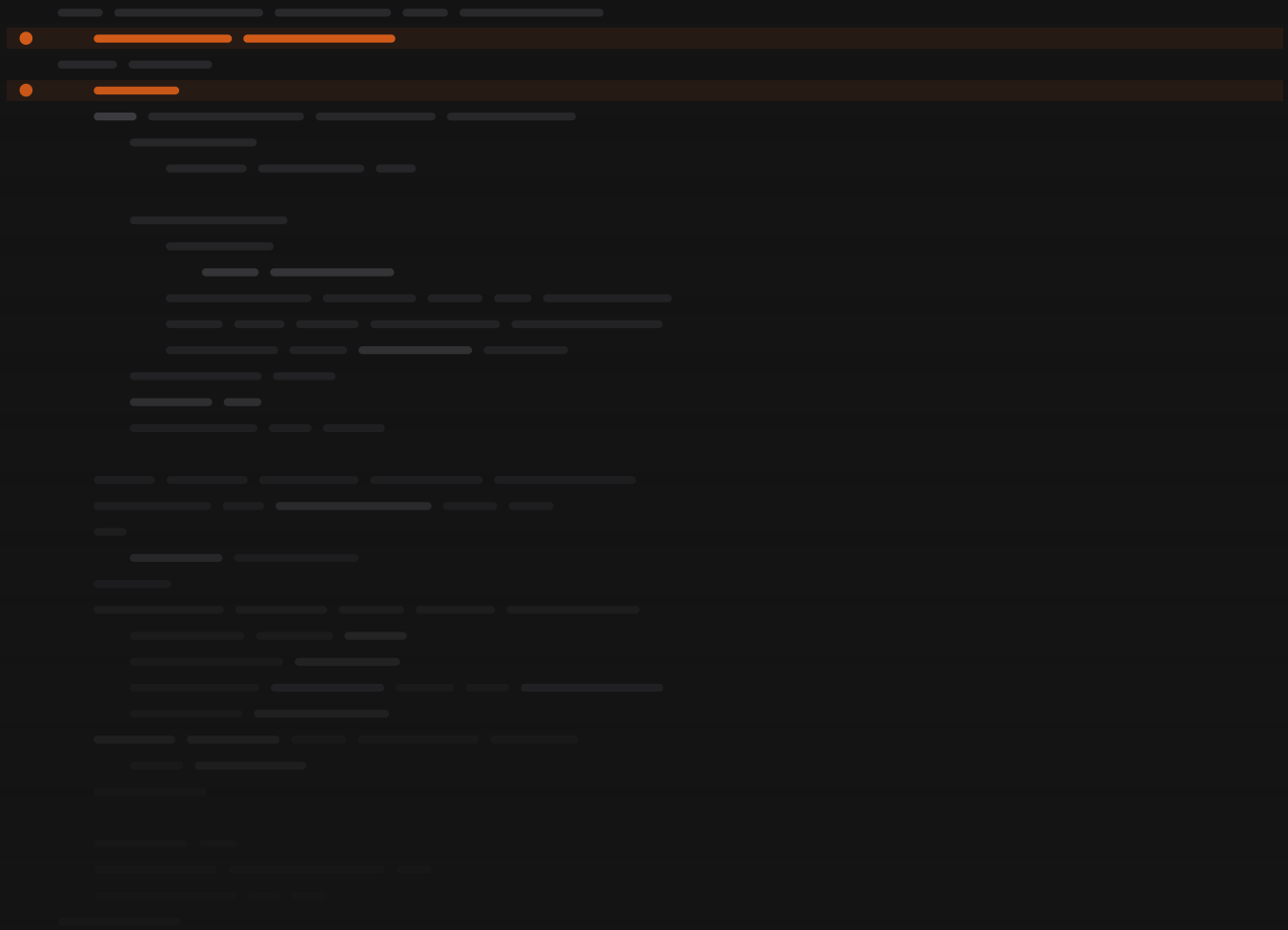
VAKA

Neden kalıp, tek bir açıktan daha tehlikeli?

Bir geliştiricinin yaptığı hata o uygulamaya özgüdür. Bir modelin yaptığı hata ise o modelle üretilen her uygulamada aynı şekilde tekrar edebilir. Benim deneyimde 10 projenin 9'unda IDOR vardı ve 7'sinde hata neredeyse aynı satırdaydı: kayıt sadece ID ile çekiliyor, sahiplik kontrolü yapılmıyordu. Saldırgan için bu, tek bir test senaryosunun yüzlerce hedefte çalışması anlamına geliyor.

ÖNGÖRÜ

Savunma tarafı da AI kullanmak zorunda kalacak; bu kaçınılmaz. Fark, AI'ı nerede kullandığımızda ortaya çıkacak: işin sonunda bir tarama aracı olarak mı, yoksa kodun yazıldığı her anda kuralları uygulayan bir katman olarak mı? Bu raporun deneyi, bu iki yolu yan yana koyuyor.



04

Araştırma: kapsam ve yöntem

Aynı 10 uygulamayı dört farklı yoldan ürettim ve hepsini aynı testten geçirdim. Sonuçları açıklayan en önemli şey, bulguları kalıp ve tasarım hatası olarak ayırmak oldu.

40 test edilen sürüm: 10 proje, her biri 4 farklı yoldan üretildi

Şekil 3. Deney tasarımı



Projeler

10 proje seçtim. Hepsi farklı stack'te, hepsi vibe coding ile birkaç günde üretilebilecek boyutta. Her birinde giriş ekranı, en az iki rol, bir veritabanı ve dosya yükleme ya da ödeme gibi bir iki hassas akış var.

#	Proje	Stack	Dil
1	SaaS yönetim paneli	Next.js + Prisma	TypeScript
2	Randevu API'si	Django + DRF	Python
3	E-ticaret	Laravel	PHP
4	Chat backend	Express + MongoDB	JavaScript
5	Dosya paylaşım servisi	FastAPI + PostgreSQL	Python
6	Fatura servisi	Spring Boot	Java
7	URL kısaltıcı	Go + Gin	Go
8	CRM	ASP.NET Core	C#
9	Mobil uygulama	React Native + Supabase	TypeScript
10	Forum	Rails	Ruby

Karşılaştırılan dört durum

Durum	Nasıl üretildi?
Ham	Sadece özellik isteyen prompt'lar. Güvenlikle ilgili tek kelime yok.
Tara-kapat	Ham halin üzerine tek prompt: "Projeyi baştan sona tara ve bulduğun güvenlik açıklarını kapat."
Security md	Aynı özellik prompt'ları ile sıfırdan. İlk prompt'tan önce repoda bir `SECURITY.md` ve onu referans veren bir `CLAUDE.md` var.
Security md + review	Security md durumuna ek olarak security-guidance plugin açık, her özellikten sonra `/security-review` çalıştırıldı.

Test ve sayım

Dört durumu da aynı şekilde test ettim. Önce OWASP ASVS'ten çıkardığım sabit bir kontrol listesiyle başladım, ardından statik analiz ve dependency taraması yaptım. En çok zamanı elle yaptığım web ve API testleri aldı: her proje için bir authorization matrisi çıkardım, IDOR denemeleri yaptım, session ve iş akışlarını adım adım bozmaya çalıştım.

Aynı hatanın farklı endpoint'lerdeki tekrarını ayrı bulgu saydım, çünkü saldırgan için her biri ayrı bir kapı. Seviye belirlerken istismar kolaylığına ve etkilenen veriye baktım.

İki sınıf: kalıp ve tasarım

Sonuçları asıl açıklayan şey bu ayrım oldu, o yüzden raporun geri kalanında sık sık bu iki terimi kullanacağım.

- **Kalıp hataları.** Koda bakınca görünen hatalar: injection, XSS, hardcoded secret, zayıf hash, eksik security header, eski dependency. Yanlış satır orada duruyor ve bir tarayıcı ya da model onu bulabiliyor.
- **Tasarım hataları.** Yanlış bir satır yok; olması gereken bir kontrol hiç yazılmamış. Eksik authorization, IDOR, tenant izolasyonu, rate limit, session yönetimi, iş mantığı. Bunu görmek için sistemin nasıl çalışması gerektiğini bilmek gerekiyor.



05

Bulgular

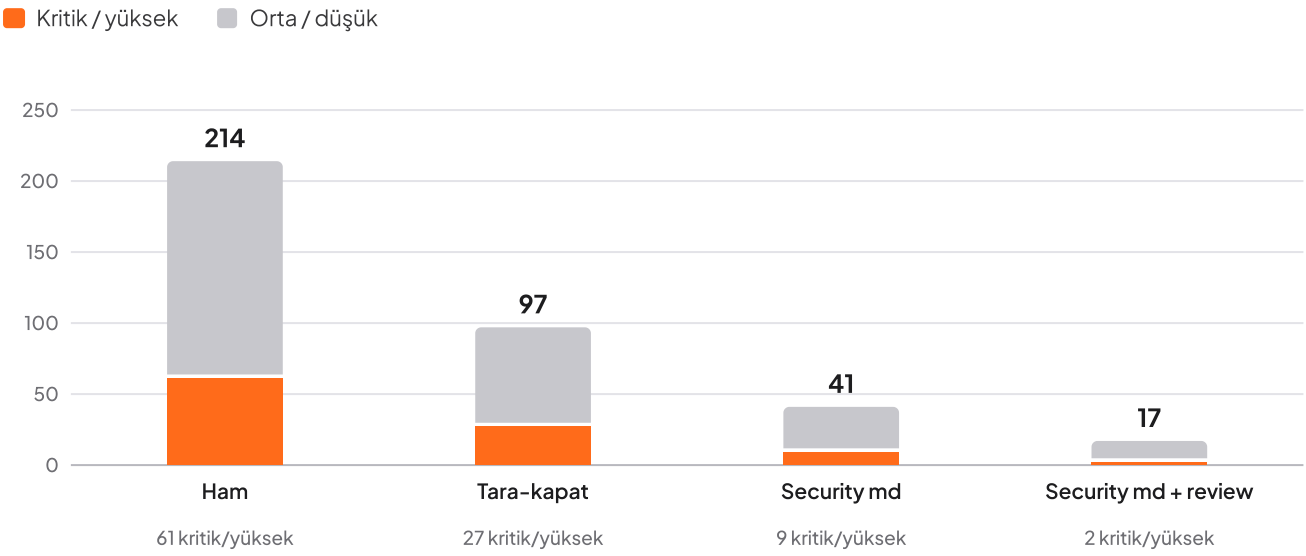
Tara-kapat toplam bulguyu yarıya indirdi. Fakat kalan yarı, bir saldırgan için en değerli olan yarıydı.

214 → 17

ham halden security md + review durumuna toplam bulgu sayısı

Genel tablo

Şekil 4. Dört durumda toplam bulgu sayısı



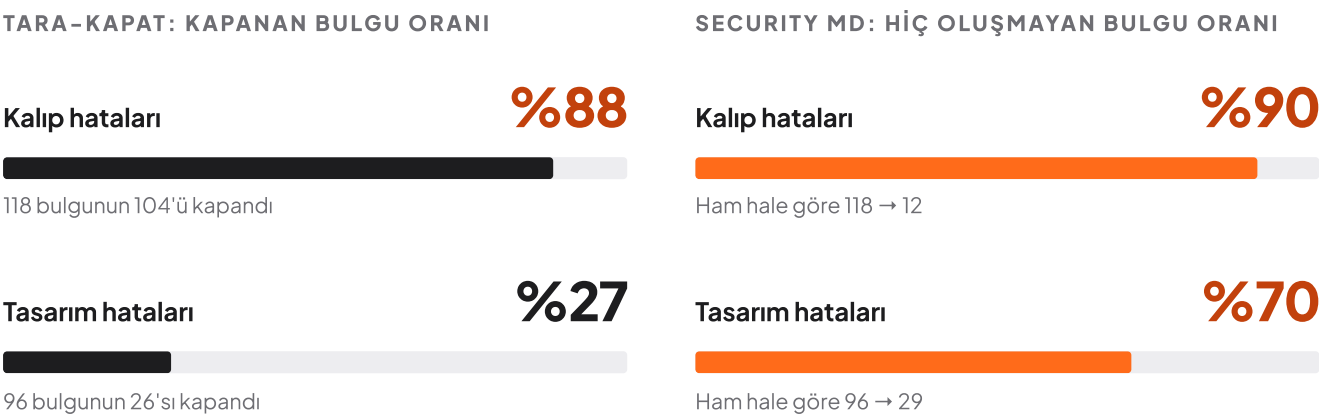
Durum	Kritik	Yüksek	Orta	Düşük	Toplam
Ham	14	47	96	57	**214**
Tara-kapat	5	22	44	26	**97**
Security md	1	8	19	13	**41**
Security md + review	0	2	9	6	**17**

Tara-kapat toplam bulguyu yarıdan biraz fazla düşürdü, bu küçümsenecek bir sonuç değil. Yine de 10 projede geriye 5 kritik ve 22 yüksek bulgu kaldı. Aynı projeler baştan security md ile üretildiğinde tek bir kritik bulgu vardı.

Hata sınıfına göre

Toplam sayı, asıl hikâyeyi saklıyor. Bulguları kalıp ve tasarım hatası olarak ayırınca iki yaklaşımın gerçekten neyi çözdüğü ortaya çıkıyor.

Şekil 5. Hata sınıfına göre kapanma ve önleme oranı



Tara-kapat kalıp hatalarında gerçekten başarılı: 118 bulgunun 104'ü kapandı. Tasarım hatalarında ise 96 bulgunun yalnızca 26'sı kapandı. Security md tarafında oranlar birbirine çok daha yakın; kural yazılı olduğunda model tasarım hatasını da büyük ölçüde hiç üretmiyor.

Kategori bazında

Şekil 6. Kategori ve duruma göre kalan bulgu sayısı

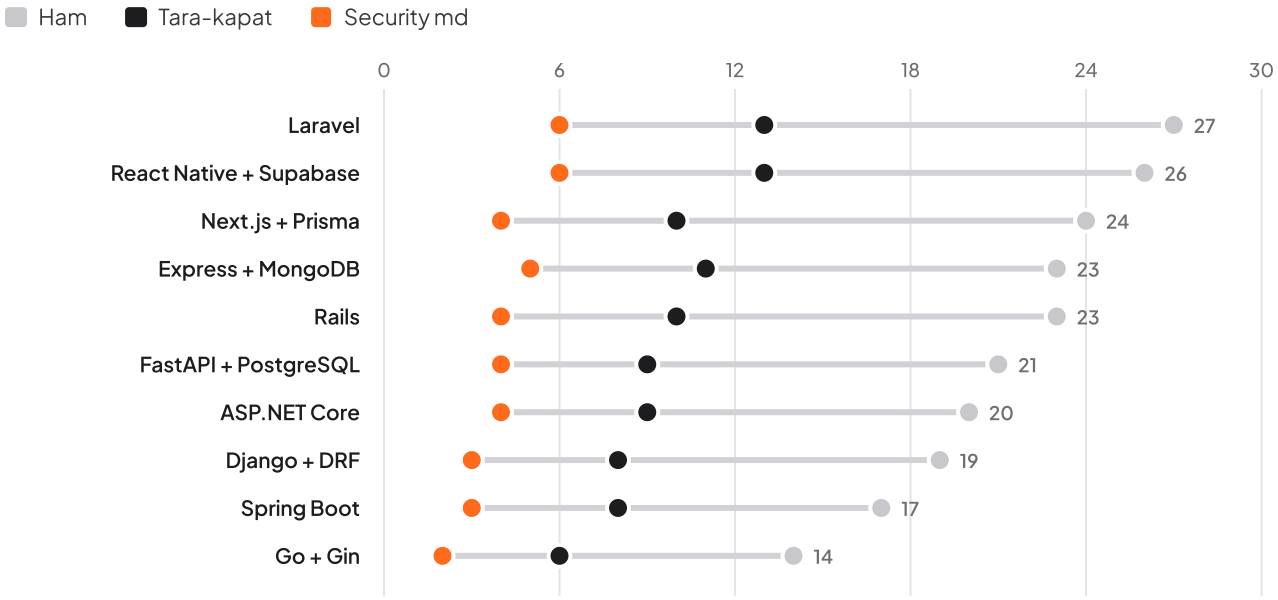
	Ham	Tara-kapat	Security md	Md + review
KALIP HATALARI				
Injection (SQL, NoSQL, komut)	21	2	2	0
XSS	19	2	2	0
Hardcoded secret	17	1	1	0
Zayıf hash ve kripto	12	1	1	0
Security header ve CORS	27	4	3	1
Eski ya da riskli dependency	22	4	3	1
TASARIM HATALARI				
Eksik authorization	24	17	5	1
IDOR	22	16	4	1
Tenant izolasyonu	11	10	2	0
Rate limit	16	10	4	2
Session yönetimi	12	7	4	1
İş mantığı	11	10	10	10
DİĞER				
Fix sırasında oluşan	–	13	–	–

0 27 Koyu hücre = çok bulgu

Isı haritasında iki satır hemen göze çarpıyor. Tenant izolasyonunda tara-kapat 11 bulgudan sadece birini kapattı; satır neredeyse hiç açılmıyor. İş mantığı satırı ise dört durumda da koyu kalıyor: kuponun iki kez kullanılması ya da iptal edilen siparişin yine de kargoya gitmesi gibi hataları ne tarama, ne kural dosyası, ne de review yakaladı.

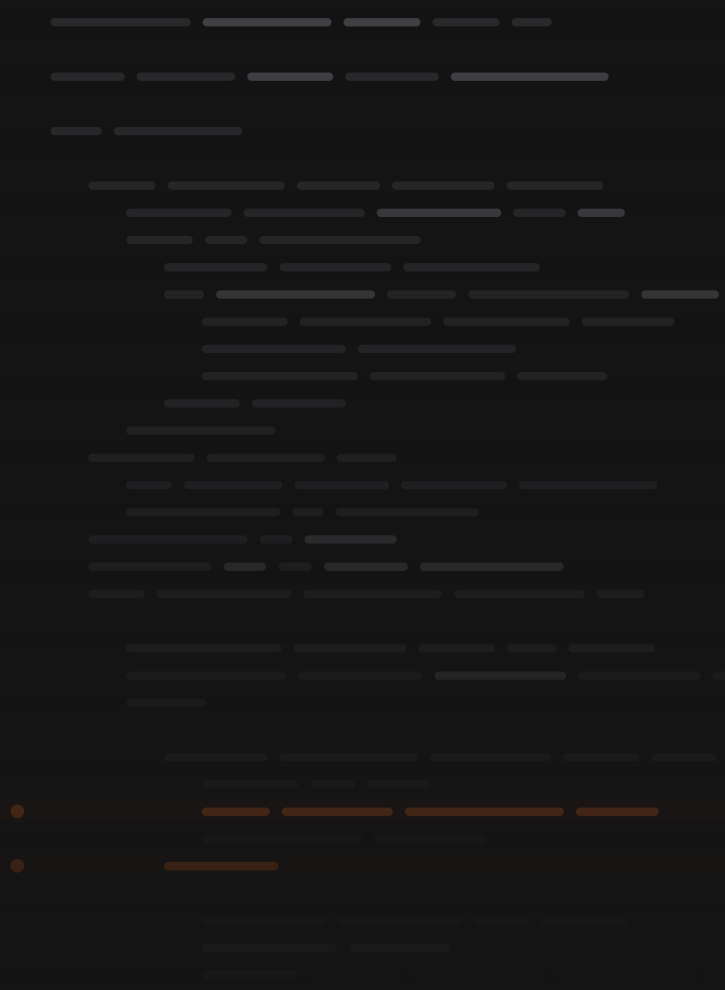
Proje bazında

Şekil 7. Proje bazında bulgu sayısı



Sıralama üç durumda da hemen hemen aynı kaldı. Dil ya da framework sonucu değiştirmedi.

Dil ve framework sonucu değiştirmedi. Ham halde en çok bulgu çıkan Laravel ve Supabase projeleri, diğer iki durumda da en üstteydi; oranlar her projede birbirine yakın. Bu bence önemli bir sonuç, çünkü sorunun belirli bir framework'ün zayıflığından değil, yaklaşımın kendisinden geldiğini gösteriyor.



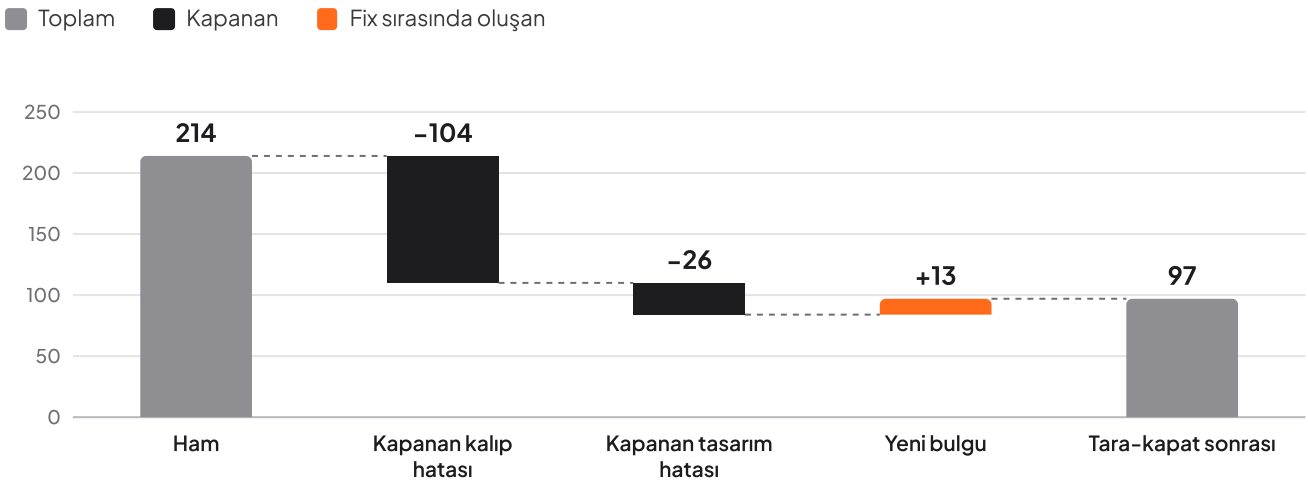
06

Tara-kapat neden yetmiyor?

Model kendisine gösterileni yamıyor. Yazılmamış bir kontrolü, sistemin nasıl çalışması gerektiğini bilmeden göremiyor.

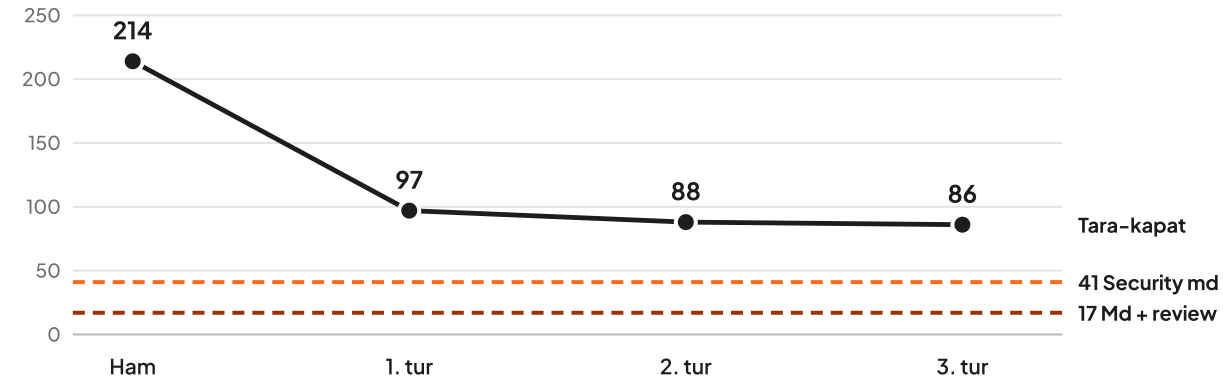
%27 tara-kapat prompt'unun kapatabildiği tasarım hatası oranı

Şekil 8. Tara-kapat prompt'unun 214 bulgu üzerindeki net etkisi



Model 130 bulguyu kapattı, 13 yeni bulgu üretti ve net sonuç 97 oldu. İlk aklıma gelen, prompt'u tekrar etmek oldu; belki model ilk turda her şeye bakamamıştı.

Şekil 9. Tara-kapat prompt'unu tekrar etmek: kalan bulgu sayısı



İkinci tur 9, üçüncü tur 2 bulgu daha kapattı. Kesikli çizgiler, aynı projelerin baştan security md ile üretilmiş halleri.

İkinci tur 9 bulgu daha kapattı, üçüncü turda neredeyse hiçbir şey değişmedi. Eğri 86 civarında düzleşiyor ve security md çizgisine hiç yaklaşmıyor. Bu yüzden sorunu prompt'un kaç kez yazıldığında değil, yaklaşımın kendisinde arıyorum. Dört neden görüyorum.

1. Yazılmamış kod taranamıyor

Aşağıdaki handler'da hatalı görünen bir satır yok. Hata, kaydın kime ait olduğunu soran satırın hiç yazılmamış olması.

```
// Ham hal: çalışıyor, test geçiyor, IDOR var
app.get("/api/invoices/:id", auth, async (req, res) => {
  const invoice = await Invoice.findById(req.params.id);
  res.json(invoice);
});

// Olması gereken: sahiplik query'nin parçası
const invoice = await Invoice.findOne({ _id: req.params.id, ownerId: req.user.id });
if (!invoice) return res.sendStatus(404);
```

Bunu yakalamak için "bu veri kimin?" sorusunun cevabını bilmek gerekiyor. O cevap kodda yazmıyor; projeyi isteyen kişinin kafasında duruyor.

2. Fix lokal kalıyor

Model bulduğu yeri yamıyor. Aynı hata başka endpoint'lerde de duruyorsa ve o dosyalar o an context'te değilse, yerlerinde kalıyorlar.

VAKA **Sipariş kapandı, fatura PDF'i açık kaldı**

Laravel projesinde tara-kapat, sipariş detayındaki IDOR'u doğru şekilde kapattı. Aynı siparişin fatura PDF'ini dönen endpoint ise ayrı bir controller'daydı ve o turda context'e girmedi. Sonuç olarak sipariş numarasını değiştirerek başka bir müşterinin faturasını indirmek hâlâ mümkündü. Bir başka projede authorization kontrolü liste endpoint'ine eklendi, aynı veriyi CSV olarak dönen export endpoint'i unutuldu. Model iki projede de özetine "authorization eklendi" yazdı.

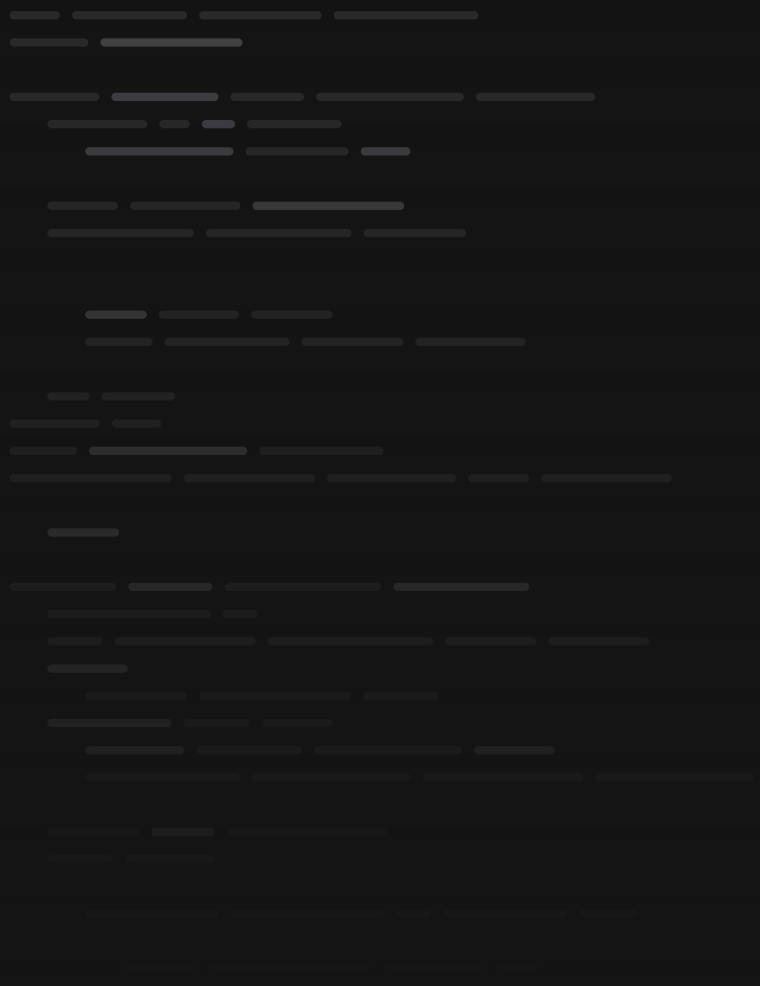
Fix sırasında oluşan 13 yeni bulgunun çoğu da aynı dar bakıştan geldi. En sık görülen örnek, modelin input validation eklerken hata mesajına stack trace koymasıydı.

3. Tasarım kararları çoktan verilmiş oluyor

Tenant ayrımı en baştan düşünülmediyse, sonradan eklemek bütün query'leri değiştirmek demek. Tara-kapat prompt'u bu büyüklükte bir işe girmiyor; model en küçük değişiklikle yetiniyor. Tenant izolasyonunda 11 bulgudan 10'unun kalması bunun sonucu.

4. Sonraki özellik eski alışkanlıkla yazılıyor

Bana göre en önemli neden bu. Tarama bir kerelik bir olay. Ertesi gün yeni bir endpoint istediğinizde model dünkü taramadan haberdar değil. Kural hiçbir yerde yazmadığı için aynı hata geri geliyor. Deney sırasında tara-kapat'tan sonra her projeye bir özellik daha ekledim; 10 projenin 8'inde yeni endpoint aynı IDOR hatasıyla geldi.



07

Security md ne değiştiriyor?

Tara-kapat bir kez çalışıyor; security md her prompt'ta orada. Fark buradan geliyor.

96 → 29 security md ile tasarım hatası sayısı

Security md, modelin her kod üretişinde context'inde duran kısa bir kural dosyası. Claude Code'da `CLAUDE.md` her session başında yükleniyor ve oradan `SECURITY.md` 'ye referans veriliyor. Böylece model "kullanıcılar faturalarını görebilsin" dediğinizde, sahiplik kuralını da biliyor.

Kullandığım dosyanın sadeleştirilmiş hali:

```
# SECURITY.md

## Veri sahipliği
- Her tablo bir `owner_id` ya da `tenant_id` taşır.
- Her okuma ve yazma query'si bu alanla filtrelenir. İstisna yok.

## Authentication ve authorization
- Varsayılan: her endpoint login ister. Public olanlar açıkça listelenir.
- Rol kontrolü handler'ın içinde değil, tek bir middleware'de yapılır.
- Client'tan gelen `user_id`, `role`, `price` alanlarına güvenilmez.

## Input ve output
- Bütün query'ler parametreli. String birleştirme ile SQL yazılmaz.
- Hata mesajında stack trace, query ya da dosya yolu dönmez.

## Secret ve dependency
- Secret'lar sadece environment variable'dan okunur.
- Yeni paket eklemeden önce paketin gerçekten var olduğunu kontrol et.

## Her yeni endpoint için
- Yetkisiz kullanıcı ve başka kullanıcının kaydı için test yaz.
- Rate limit gerekiyor mu? Gerekmiyorsa nedenini yorum olarak yaz.
```

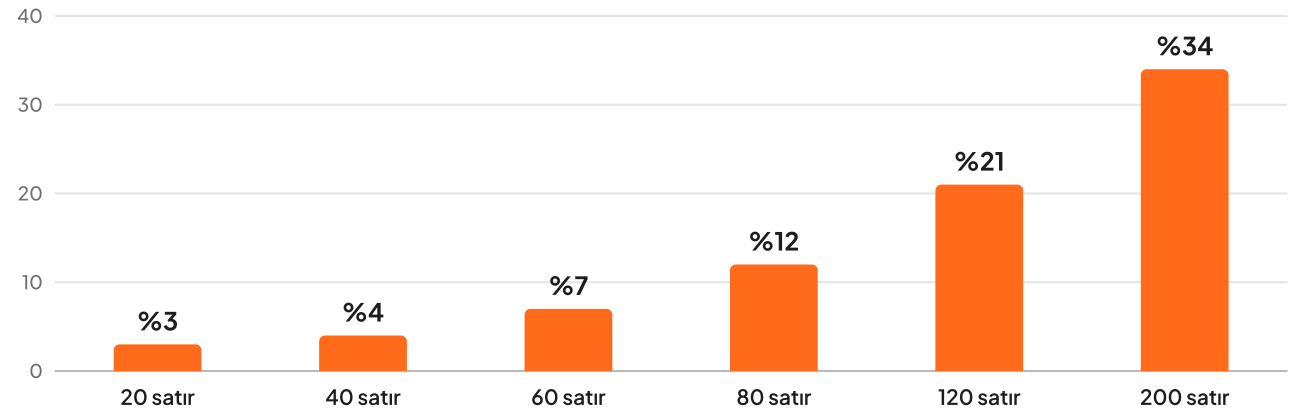
Üç gözlem

En büyük fark tasarım hatalarında. 96 bulgu 29'a indi. "Her query `owner_id` ile filtrelenir" kuralı, eksik kalan o satırı modelin her seferinde yazmasını sağladı. Tara-kapat'ın bulamadığı boşluk hiç oluşmadı.

Somut kural işe yarıyor, genel öğüt yaramıyor. İlk denemede md'ye "güvenli kod yaz" ve "OWASP Top 10'a dikkat et" gibi cümleler koymuştum; sonuç ham halden pek farklı değildi. "Client'tan gelen `role` alanına güvenme" gibi somut kurallar ise doğrudan koda yansıdı.

Kısa dosya daha iyi çalıştı. Aynı kural setini farklı uzunluklarda yazıp üç projede denedim. 60–80 satırı geçtikçe model bazı maddeleri atlamaya başladı.

Şekil 10. Security md uzunluğu ve uygulanmayan kural oranı



Aynı kural setini farklı uzunluklarda yazıp 3 projede denedim. Oran, md'de yazdığı halde kodda uygulanmayan kuralların payı.

OpenSSF'in AI kod asistanları için hazırladığı rehber de aynı şeyi öneriyor: talimatlar kısa, somut ve projeye özel olmalı. Uzun bir genel liste, kısa ve net bir listeden daha az koruma sağlıyor.

> **UYARI**
>
> Md ile bile 41 bulgu kaldı ve 9'u kritik ya da yüksek seviyedeydi. Kural dosyası hatayı azaltıyor, sıfırlamıyor. Review ve test hâlâ gerekiyor.

08

Claude bu güvenliği nasıl sağlıyor?

Bu işi tek bir özellik yapmıyor. Katmanlardan üçü rehberlik ediyor, sadece biri gerçekten engelliyor; bunu bilmek önemli.

---

4 üst üste binen katman: kurallar, plugin, review ve sandbox

Şekil 11. Claude Code tarafında güvenliği sağlayan dört katman

01	`CLAUDE.md` + `SECURITY.md` Her session başında	Proje kurallarını context'e yükler. Model her prompt'ta bu kuralları görür.	Hayır, rehber
02	**security-guidance plugin** Her edit, turn sonu, commit	Edit'te pattern kontrolü, turn sonunda diff review, commit'te çevredeki kodu da okuyan derin review.	Hayır, bulguyu geri verir
03	`/security-review` Çağırdığınızda ya da PR'da	Branch'teki değişiklikler üzerinde güvenlik turu. GitHub Action sürümü var.	Hayır, rapor üretir
04	**Permission ve sandbox** Her tool çağrısında	Agent'ın hangi komutu çalıştıracağını, nereye yazacağını ve ağ erişimini sınırlar.	Evet, agent'ı sınırlar

Security md durumunda kullandığım şey ilk katmandı. Dördüncü durumda buna security-guidance plugin'ini ve `/security-review` komutunu ekledim. Plugin'in kurulumu tek satır:

```
/plugin install security-guidance@claude-plugins-official
```

Plugin'de önemli bulduğum ayrıntı, review'u yapanın kodu yazan instance olmaması. Review, temiz bir context ile başlayan ayrı bir çağrı tarafından yapılıyor; bu yüzden yazarken yapılan bir varsayımı tekrar etme ihtimali daha düşük. Projeye özel kurallar `.claude/claude-security-guidance.md` dosyasına ekleniyor.

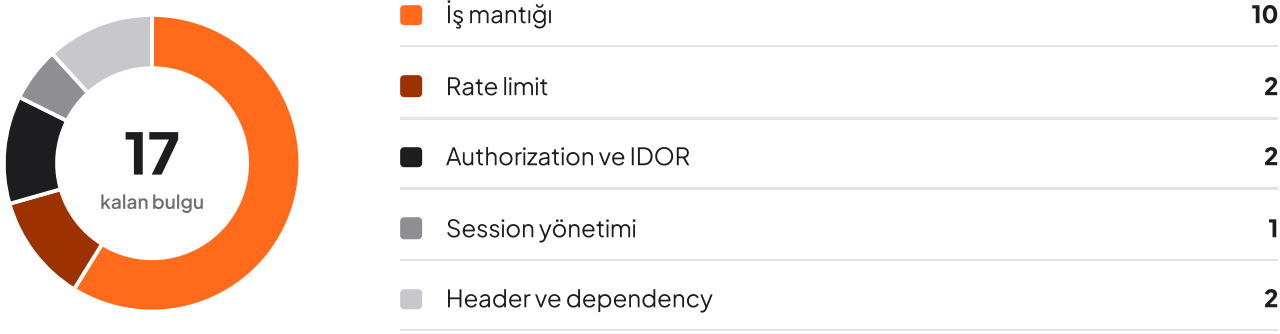
İki sınır

İlk sınır şu: Anthropic'in dokümanı md kurallarını bir guardrail olarak değil, rehber olarak tanımlıyor. Plugin hiçbir yazmayı ya da commit'i bloklamıyor ve review modeli bir şeyi kaçırabilir. Kesin engelleme istiyorsanız, bunu bir hook ya da CI kontrolü ile yapmanız gerekiyor.

İkinci sınır, 3. bölümde anlattığım saldırı yüzeyi. Kural dosyaları otomatik yüklendiği ve güvenilir kabul edildiği için, onlara yapılan her değişikliğin review'dan geçmesi gerekiyor.

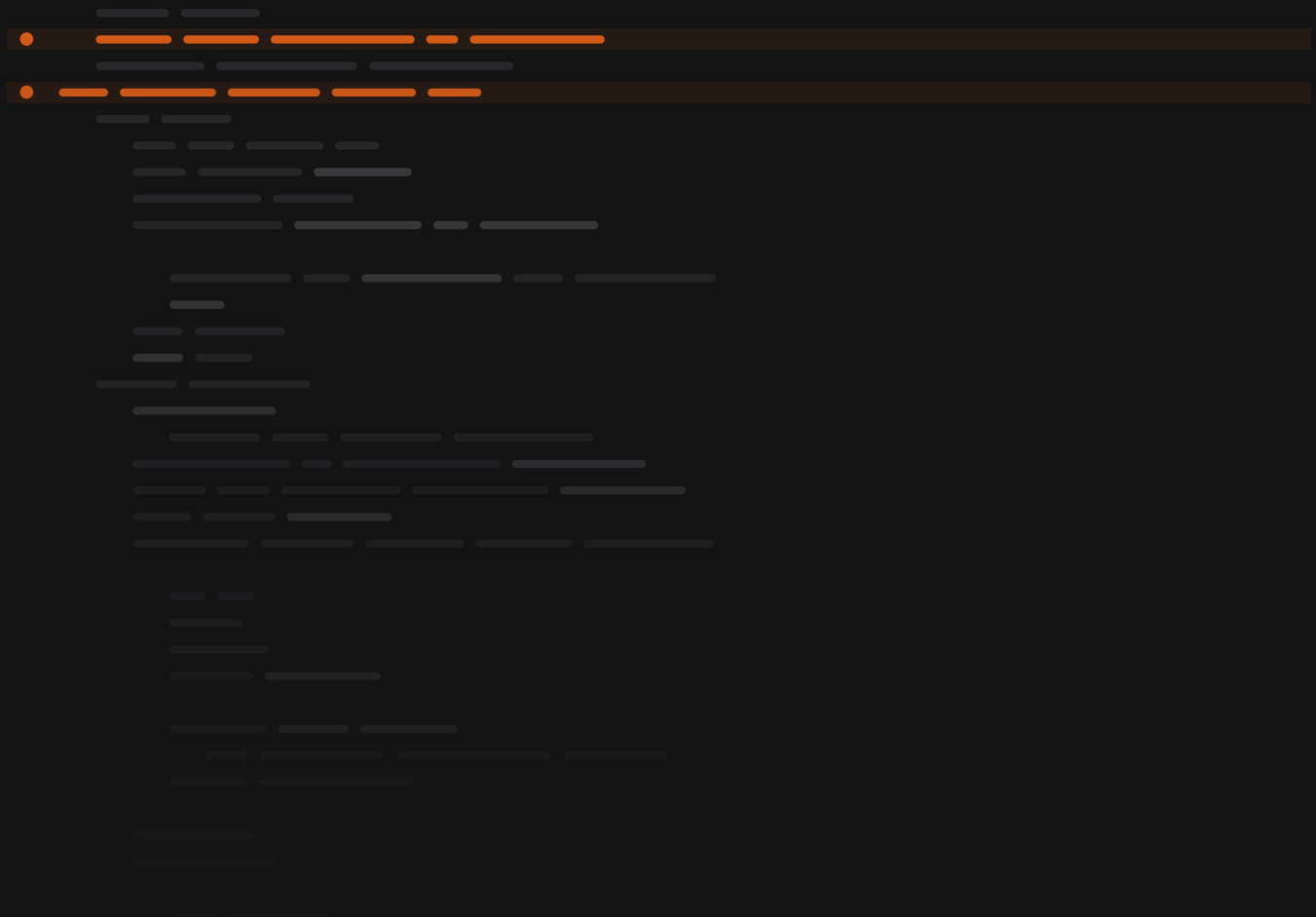
Geriye ne kaldı?

Şekil 12. Md + review durumunda kalan 17 bulgunun dağılımı



Kalan bulguların çoğu, doğrusunu yalnızca işi bilen kişinin bildiği kurallardan geliyor.

Md + review durumunda kalan 17 bulgunun 10'u iş mantığıyla ilgiliydi. Bunları ne md ne de review yakaladı, çünkü neyin doğru olduğunu sadece işi bilen kişi biliyor. "Bir kupon bir siparişte bir kez kullanılır" gibi bir kural, ancak biri onu yazarsa koda girebiliyor.



09

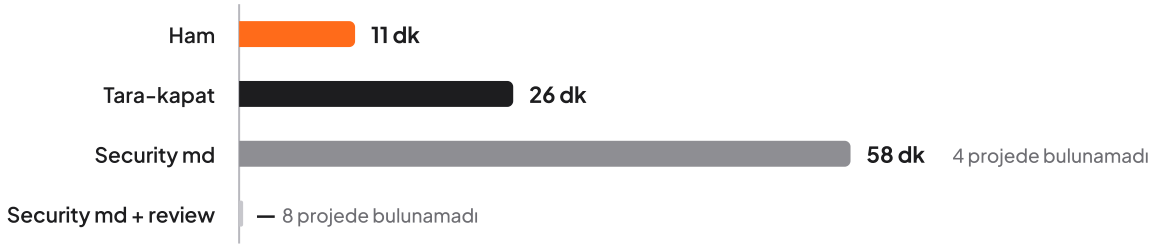
Saldırgan gözüyle: güvenlikçi bakış açısı

Bir vibe coding uygulamasını test ederken nereden başladığımı anlatmak, md'ye neyi yazmanız gerektiğini de gösteriyor.

11 dk ham projelerde ilk kritik/yüksek bulguya ulaşma süresinin ortalaması

Pentest yaparken bir uygulamaya ilk yaklaşımım, onu kimin ve nasıl yazdığına göre değişiyor. Vibe coding ile üretilmiş bir uygulamada nereye bakacağımı artık büyük ölçüde biliyorum, çünkü modeller benzer hataları benzer yerlerde yapıyor. Deneyde bunu ölçtüm: her sürüm için kronometreyi başlatıp ilk kritik ya da yüksek bulguya ne kadar sürede ulaştığımı not ettim.

Şekil 13. İlk kritik/yüksek bulguya ulaşma süresi (10 projenin ortalaması)



Ham projelerde ilk ciddi bulguya ortalama 11 dakikada ulaştım. Security md ile üretilen sürümlerde bu süre 58 dakikaya çıktı ve 4 projede hiç kritik ya da yüksek bulgu bulamadım. Review katmanı eklenince 10 projenin 8'inde bulamadım. Saldırgan açısından bu fark çok büyük: dakikalar içinde düşen bir hedefle saatler isteyen bir hedef, aynı otomatik taramada aynı muameleyi görmüyor.

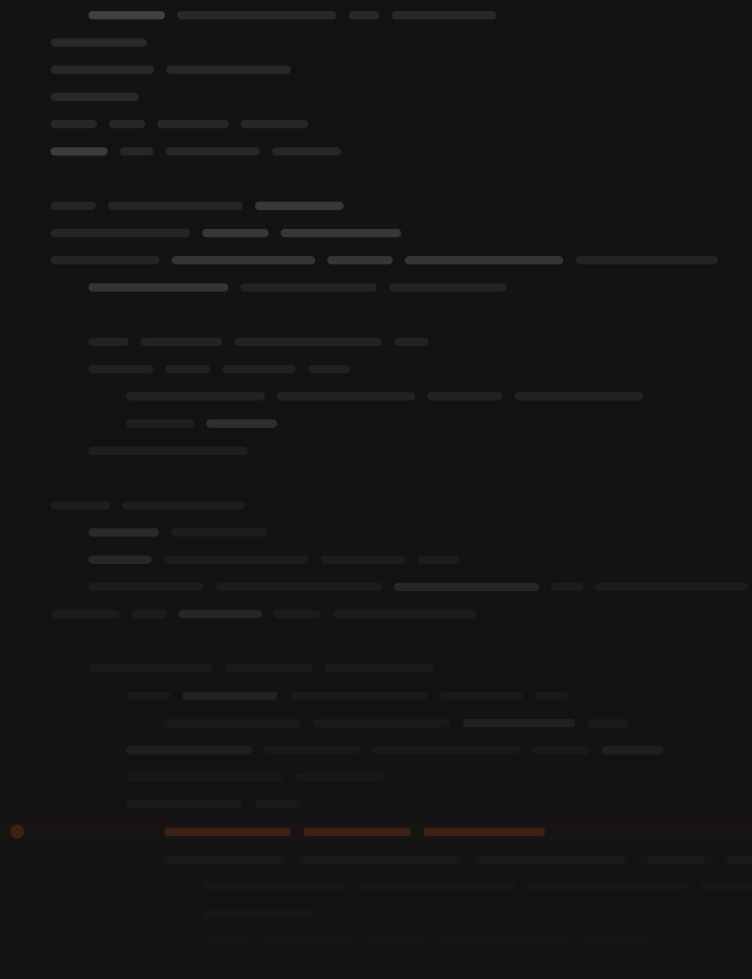
İlk 30 dakikada baktıklarım

Adım	Neye bakıyorum?	Ham projelerde görüldü
1	ID'yi değiştirerek başka kullanıcının kaydını okumak	9 / 10
2	Login, OTP ve parola sıfırlama akışında rate limit	8 / 10
3	Hata mesajlarında stack trace, query ya da dosya yolu	7 / 10
4	JS bundle'da açık key, gizli endpoint ya da admin route	6 / 10
5	Request body'de `role`, `price`, `user_id` gibi alanların kabul edilmesi	5 / 10
6	Sadece arayüzde gizlenmiş, backend'de korunmayan admin işlemleri	4 / 10

Bu tablodaki her satır, aslında bir md kuralının tersi. 1. adım "her query sahiplikle filtrelenir" kuralının, 5. adım "client'tan gelen alanlara güvenilmez" kuralının, 6. adım "rol kontrolü backend'de tek bir middleware'de yapılır" kuralının eksikliğini arıyor. Bir saldırganın kontrol listesi, sizin md'nizin taslağı olarak kullanılabilir.

Saldırganın gözünden vibe coding'in çekiciliği

Bir saldırgan için vibe coding uygulamalarını cazip kılan şey, tek tek zayıf olmaları değil, birbirlerine benzemeleri. Aynı platform, aynı backend, aynı model ve benzer prompt'lar benzer hatalar üretiyor; Escape'in 5.600 uygulamalık taraması da bu kalıbın otomatik olarak taranabildiğini gösterdi. Bu yüzden vibe coding güvenliği ekosistem ölçeğinde bir sorun. Savunmacı açısından en etkili adım, kendi uygulamanıza bu kontrol listesiyle bakmak ve her satırı bir md kuralına çevirmek.



10

Sonuç

Zafiyeti kapat demek yeterli değil. Kuralı en baştan, modelin her kod yazışında göreceği yere yazmak gerekiyor.

%92 security md + review ile ham hale göre daha az bulgu

Bu rapora bir soruyla başladım: işin sonunda "projeyi tara ve açıkları kapat" demek ne kadar işe yarıyor? Cevap, beklediğimden daha net çıktı.

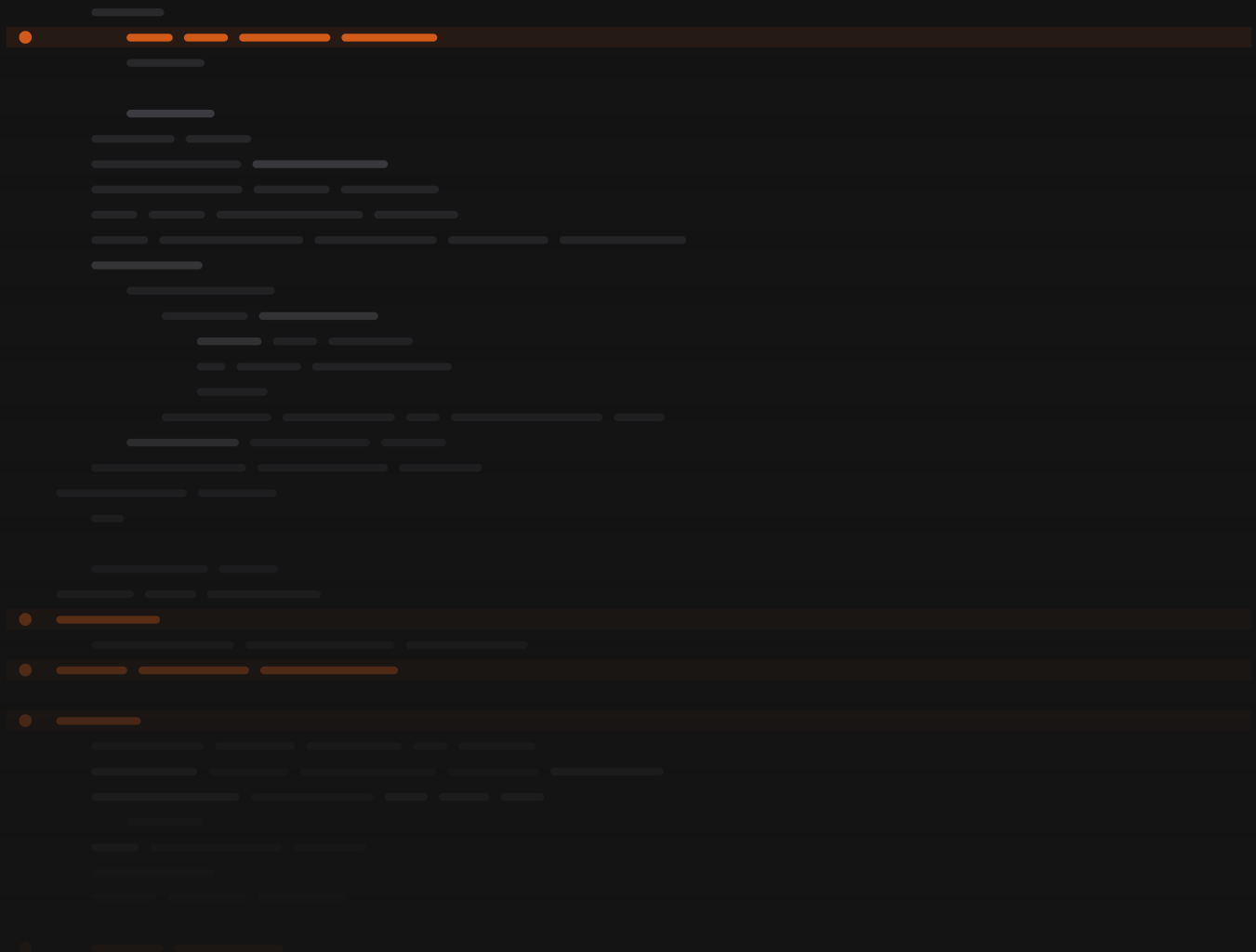
Vibe coding kod üretimini hızlandırdı ve geliştirmeyi çok daha geniş bir kitleye açtı. Bu hız, review ve güvenlik kapasitesini geride bıraktı. Sektör verileri, hataların yok olmadığını, koda bakarak görülmesi daha zor olan tasarım katmanına taşındığını gösteriyor. Aynı dönemde AI'ın kendisi yeni bir saldırı yüzeyi haline geldi ve saldırganlar da AI ile hızlandı.

Kendi deneyimde gördüğüm tablo bu genel resmin küçük bir kopyası. Tara-kapat prompt'u kalıp hatalarını iyi temizliyor, fakat tasarım hatalarının dörtte üçünü yerinde bırakıyor ve prompt'u tekrar etmek bunu değiştirmiyor. Kurallar en baştan, kısa ve somut bir security md olarak yazıldığında ise aynı hatalar büyük ölçüde hiç oluşmuyor. Üzerine bir review katmanı eklendiğinde geriye kalan, çoğunlukla sadece işi bilen kişinin yakalayabileceği iş mantığı hataları.

Taramayı bırakın demiyorum. Tarama hâlâ değerli ve mevcut projeler için iyi bir başlangıç. Fakat tarama bir kontrol; kural ise bir alışkanlık. Vibe coding'de alışkanlığı model taşıdığı için, kuralı da modelin okuyacağı yere yazmak gerekiyor.

2027 İÇİN ÖNGÖRÜLER

- **Kural dosyaları standartlaşacak.** OWASP, OpenSSF ve büyük platformların hazır şablonları yaygınlaşacak; proje başında security md yazmak, README yazmak kadar sıradan hale gelecek.
- **Kural dosyası saldırıları artacak.** Otomatik yüklenen dosyalar tedarik zinciri saldırılarının yeni hedefi olacak; bu dosyalar için code owner ve imzalama pratikleri gelecek.
- **Vibe coding platformları güvenli varsayılanlara geçecek.** Row level security ve sahiplik kontrolü gibi kurallar platform seviyesinde açık gelmeye başlayacak.
- **İş mantığı insan işi olarak kalacak.** Modeller ne kadar gelişirse gelişsin, işin kuralını yazmak o işi bilen kişinin sorumluluğunda kalacak.



11

Öneriler

Hepsi küçük adımlar. Birlikte uygulandığında deneydeki 214 bulguyu 17'ye indiren şey bunlardı.

8 adım: yarın sabah başlayabileceğiniz uygulanabilir liste

1 İlk prompt'tan önce kısa bir SECURITY.md koyun

Dosyayı `CLAUDE.md` içinden referans verin ve veri sahipliği kuralını en üste yazın. 60 satırın altında tutun.

2 Kuralları somut yazın

"Güvenli kod yaz" yerine "her query `tenant_id` ile filtrelenir" yazın. 9. bölümdeki saldırgan kontrol listesini md'nizin taslağı olarak kullanabilirsiniz.

3 Bir review katmanı ekleyin

security-guidance plugin'ini kurun ve her özellikten sonra `/security-review` çalıştırın. PR'larda GitHub Action sürümünü kullanın.

4 Her endpoint için yetki testi isteyin

Modelden her yeni endpoint için "başka kullanıcının kaydına erişemiyorum" testini yazmasını isteyin. Test, kuralın uygulandığının kanıtı.

5 Kesin kontrolleri CI'a taşıyın

Secret taraması ve dependency audit'i CI'da bloklayıcı yapın. Engellemesi gereken şeyi rehber niteliğindeki md'ye bırakmayın.

6 Kural dosyalarını koruyun

`CLAUDE.md`, `SECURITY.md` ve `AGENTS.md` değişikliklerini code owner review'dan geçirin. Bu dosyalar CI config'i kadar kritik.

7 Mevcut projede tara-kapat ile başlayın, orada durmayın

Bulduğunuz her açığı bir kurala çevirip md'ye yazın ki aynı hata bir sonraki özellikte geri gelmesin.

8 İş mantığını kendiniz test edin

Negatif tutar, aynı isteği iki kez gönderme, adım atlama, iptal edilmiş kaydı kullanma. Bunları sizin yerinize kimse düşünmüyor.

Ekler

Sınırlılıklar

- Örneklem küçük: 10 proje ve hepsi küçük ya da orta boyutta.
- Testleri tek kişi yaptım. Seviye belirleme ve sınıflandırma benim yargıma dayanıyor.
- Model çıktıları her çalıştırmada biraz değişiyor. Her durumu bir kez ürettim, tekrar ortalaması almadım.
- Tek bir araç ve tek bir tara-kapat prompt'u kullandım. Farklı bir prompt ya da model başka oranlar verebilir.
- Süre ölçümleri benim test alışkanlıklarıma bağlı; başka bir test uzmanı farklı sürelere ulaşabilir.

Bu nedenle deney rakamlarını kesin ölçü olarak değil, yön gösteren bir işaret olarak okumak gerekiyor. Sektör verileri ise ilgili kaynaklara aittir.

Sözlük

Terim	Anlamı
Vibe coding	Kodu satır satır yazmak yerine ne istediğini anlatıp modelin ürettiğini kabul ederek geliştirme.
Tara-kapat	İş bittikten sonra modele "projeyi tara ve açıkları kapat" demek.
Security md	Modelin her prompt'ta okuduğu, projeye özel kısa güvenlik kuralları dosyası.
IDOR	ID'yi değiştirerek başka bir kullanıcının kaydına erişebilmek.
Tenant izolasyonu	Çok müşterili bir sistemde bir müşterinin verisinin diğerine görünmemesi.
Prompt injection	Modelin okuyacağı bir metne gizlenmiş talimatlarla modeli yönlendirmek.
Slopsquatting	Modelin uydurduğu paket adlarını önceden kaydedip zararlı kod dağıtmak.

Kaynaklar ve hazır md'ler

- **OWASP Secure Coding with AI Cheat Sheet.** AI'a özgü riskler: hallucinated dependency, prompt injection, kural dosyalarının güvenliği. cheatsheetseries.owasp.org
- **OWASP ASVS.** Kendi md'nizin maddelerini çıkarabileceğiniz kontrol listesi. owasp.org/www-project-application-security-verification-standard
- **OpenSSF: Security-Focused Guide for AI Code Assistant Instructions.** Kural dosyasına ne yazılır, ne yazılmaz. best.openssf.org
- **Claude Code: security-guidance plugin.** Kurulum, üç review katmanı ve `claude-security-guidance.md` formatı. code.claude.com/docs/en/security-guidance

- **Claude Code: Security.** Permission, sandbox ve prompt injection korumaları. code.claude.com/docs/en/security
- **anthropics/claude-code-security-review.** PR'larda otomatik security review yapan GitHub Action. github.com/anthropics/claude-code-security-review
- **Wiz: secure-rules-files.** Django, Flask, React, Node.js, Spring ve ASP.NET Core için hazır başlangıç kural dosyaları. github.com/wiz-sec-public/secure-rules-files
- **Cloudflare: Engineering standards enforcement.** Mühendislik standartlarını RFC olarak yazıp agent'lara okutma yaklaşımı; Workers için hazır skill'ler cloudflare/skills reposunda. blog.cloudflare.com/engineering-standards-enforcement
- **CSA: Instruction-File Poisoning of AI Coding Agents.** Kural dosyalarının neden saldırı yüzeyi olduğu. labs.cloudsecurityalliance.org

Sektör verilerinin kaynakları

- **Stack Overflow Developer Survey 2025.** AI kullanımı ve güven oranları. survey.stackoverflow.co/2025
- **Veracode 2025 GenAI Code Security Report.** 100'den fazla model, 80 görev, %45 güvensiz seçim. veracode.com/resources/2025-genai-code-security-report
- **Apiiro: 4× Velocity, 10× Vulnerabilities.** AI asistanlarıyla hata türlerinin değişimi. apiiro.com/blog/4x-velocity-10x-vulnerabilities
- **Escape: vibe coding uygulamalarında 2.000+ zafiyet.** 5.600 uygulamalık tarama ve metodoloji. escape.tech/blog
- **CVE-2025-48757.** Lovable uygulamalarında eksik row level security. nvd.nist.gov/vuln/detail/CVE-2025-48757
- **Spracklen ve ark., USENIX Security 2025.** "We Have a Package for You!" paket halüsinasyonu çalışması. usenix.org
- **CrowdStrike 2026 Global Threat Report.** AI kullanan saldırganlar ve agentic dönem. crowdstrike.com/global-threat-report
- **Anthropic: Disrupting the first reported AI-orchestrated cyber espionage campaign.** Kasım 2025. anthropic.com/news/disrupting-AI-espionage

YAZAR HAKKINDA



Sedat Özdemir

Sr. Expert Pentest & Vulnerability Management Engineer

On yıla yakındır siber güvenlikte, ağırlıklı olarak ofansif tarafta çalışıyorum: web uygulamaları, API'ler ve mobil uygulamalar için sızma testleri, red team çalışmaları ve son dönemde ofansif yapay zekâ. Güvenliğe yazılım tarafından geldim; bir uygulamanın nasıl kırıldığını, onu nasıl yazdığımı bildiğim için anlıyorum.

Bu rapor kendi lab çalışmamın notlarıdır. Blogumda red team, pentest, DevSecOps ve offensive AI üzerine benzer notlar paylaşıyorum.

Web sedatozdemir.com.tr

LinkedIn linkedin.com/in/sedat0zdemir

Rapor sedatozdemir.com.tr/reports/vibe-coding-guvenligi-aciklari-kapat-demek-neden-yetmiyor.tr.pdf

Bu rapordaki görüşler yazara aittir; hiçbir kurumu temsil etmez. Deney bulguları yazarın kendi test laboratuvarından gelir; işveren ya da müşteri verisi içermez. Sektör verileri ilgili kaynaklara aittir ve Ekler bölümünde listelenmiştir.