Konuşma Metni

Slayt 2

“Herkese merhaba, ben Çağatay Üresin. Bugün yazılım güvenliği tarafında çok kritik bir konu olan statik kod analizi ve bunun gerçek hayatta nasıl kullanıldığını SonarQube üzerinden anlatacağım.

Klasik dünyada yazılım geliştirme genelde ‘yaz – derle – deploy et – sonra sorun çıkarsa bakarız’ şeklinde ilerliyor. Bu yaklaşımda güvenlik çoğu zaman en sona bırakılıyor.

DevOps kavramıyla birlikte ‘hızlı ve sürekli teslimat’ hayatımıza girdi; ama hızlandıkça yaptığımız hataları da daha hızlı şekilde production’a taşımaya başladık. İşte tam burada DevSecOps devreye giriyor: Güvenliği, geliştirmenin ve operasyonun tam ortasına yerleştiriyor.

Ekranda gördüğünüz diyagramda solda geleneksel akışı görüyorsunuz: Code → Build → Deploy → Production. Sağda ise DevSecOps yaklaşımı var; burada Build aşamasına gelmeden önce Static Code Analysis (SAST) dediğimiz bir kontrol katmanı ekleniyor.

Bu sunumun hedefi, statik kod analizini teorik olarak anlatmanın ötesinde, kendi kurduğum SonarQube sistemi üzerinde, GitHub Actions, Quality Gate ve IDE entegrasyonlarıyla uçtan uca bir örnek göstermek olacak.”

Slayt 3

“DevOps aslında çok karışık bir şey değil; kısaca ‘Development + Operations’. Yani yazılım geliştirme ekipleri ile sistem/operasyon ekiplerinin birlikte, otomatik ve sürekli çalışan bir teslimat hattı kurması.

Ekrandaki diyagramda klasik DevOps döngüsünü görüyorsunuz: Plan → Code → Build → Test → Release → Deploy → Operate → Monitor → tekrar Plan. Bu döngüde CI yani Sürekli Entegrasyon; kodun birleştirildiği, testlerin otomatik koştuğu kısım. CD ise bu işin deploy tarafı.

Fakat bu güzel DevOps dünyasının bir eksiği var: Güvenlik. Çoğu zaman güvenlik ya manuel kod incelemesiyle, ya da üretimde bir zafiyet ortaya çıktığında gündeme geliyor. Yani güvenlik, sürecin ortasında değil, kenarında duruyor. İşte bu noktadan sonra DevSecOps kavramı ortaya çıkıyor.”

Slayt 4

“DevSecOps, DevOps’un güvenlik ile evrilmiş hali gibi düşünebiliriz. Yani artık sadece geliştirme ve operasyon değil; güvenlik de bu ekibin birinci sınıf vatandaşı oluyor.

Slaytta iki ayrı pipeline görüyorsunuz. Üstte klasik DevOps: Code → Build → Test → Deploy. Altta ise DevSecOps pipeline’ı var; burada özellikle iki yeni katman dikkat çekiyor:

Birincisi SAST – Static Application Security Testing, yani statik kod analizi. Kod daha derlenmeden, çalışmadan; sadece kaynak kod üzerinden güvenlik ve kalite analizi yapılması. İkincisi ise güvenlik testlerinin pipeline içine gömülmüş olması.

Buradaki kilit kavram ‘Shift Left Security’. Yani güvenlik kontrollerini, mümkün olduğunca sürecin en başına, kod yazımına doğru kaydırmak. Ne kadar erken yakalarsak, bir hatayı düzeltmenin maliyeti o kadar düşük oluyor.

Bu sunumda göstereceğim SonarQube, tam olarak bu diyagramdaki SAST kutusunun içini dolduruyor. Geliştirici kodunu push ettiği anda, pipeline'da otomatik olarak analiz devreye giriyor."

Slayt 5

"Statik kod analizi, aslında ismi kadar karmaşık değil. Kodu çalıştırmadan, sadece kaynak kod üzerinden yapılan analiz. Yani 'programı çalıştır, saldırı yap, sonucu ölç' mantığından farklı; burada kodun kendisine bakıyoruz.

Ekrandaki diyagramda, bizim klasik CI pipeline'ımızı görüyorsunuz: Geliştirici kodunu GitHub'a push ediyor, ardından GitHub Actions devreye giriyor. Bu noktada SonarQube ile bir SAST adımı ekliyoruz. Bu adımda kod taranıyor, bug'lar, güvenlik açıkları ve kötü kokular (code smells) aranıyor.

Taramadan sonra Quality Gate devreye giriyor. Bu da 'Bu projeyi geçireyim mi, yoksa build'i durdurayım mı?' diyen bir trafik ışığı gibi. Eğer Quality Gate başarısızsa, pipeline'ı kırmızıya çekip build'i durduruyoruz. Eğer geçerse, normal build, test ve deploy süreçleri devam ediyor.

Bu yaklaşımın en büyük avantajı, hataları build öncesinde yakalamamız. Yani problem daha production'a gitmeden, hatta test ortamına bile gelmeden engelleniyor. Bu hem güvenlik riskini azaltıyor, hem de uzun vadede ciddi bir zaman ve maliyet kazancı sağlıyor.

Birazdan canlı demoda tam olarak bu akışı – GitHub Actions + SonarQube + Quality Gate – çalışırken göstereceğim."

Slayt 6

"Artık SAST'in ve DevSecOps'un resmini kabaca gördük. Şimdi bu resimde SonarQube'ün tam olarak nereye oturduğuna bakalım.

Ekrandaki diyagramda ortada bir SonarQube Server görüyorsunuz. Geliştirici tarafta iki ayrı kanal var:

Birincisi IDE tarafı. VSCode veya başka bir ortamda çalışan SonarLint eklentisi, SonarQube'ün kurallarını geliştiricinin masasının üzerine getiriyor. Yani daha kodu push etmeden, inline uyarılarla 'Bu kodda güvenlik açığı var, bu kodda kötü bir koku var' diyebiliyor.

İkincisi GitHub tarafı. Geliştirici kodu push ettiğinde GitHub Actions devreye giriyor, SonarScanner'ı çalıştırıyor ve kodu SonarQube Server'a analiz için gönderiyor.

SonarQube bu analiz sonucunda bug, vulnerability, code smell ve security hotspot gibi issue'ları üretiyor, bunları web arayüzünde gösteriyor, Quality Gate kararı veriyor ve istersek projeye badge olarak yansıtıyor.

Bu sunumda;

GitHub Actions ile otomatik taramayı,

Quality Gate'in build'i nasıl durdurabildiğini,

Her bir issue türünden örnekleri,

Ve IDE tarafında SonarLint'in aynı sorunları nasıl gösterdiğini canlı olarak göreceğiz.

Yani aslında DevSecOps zincirini teoride değil, uçtan uca çalışan bir örnek üstünden birlikte izleyeceğiz."

Slayt 8

“SonarQube mimarisi aslında üç parçadan oluşuyor: Scanner, Server ve Database. Geliştirici kodu push ettiği anda GitHub Actions üzerinden SonarScanner tetikleniyor. Scanner kodu analiz ediyor ve tüm bulguları SonarQube Server’a gönderiyor.

Server bu veriyi işler, Quality Gate kararını verir ve sonuçları dashboard’da saklar.

Burada önemli bir nokta şu: Scanner sadece veri toplayan taraf; analiz motoru server’da çalışıyor. UI ise sadece gösterim değil; aynı zamanda Issue yönetimi, kullanıcı atama ve tarihsel metriklerin tutulduğu bir yönetim paneli.”

Slayt 9 - 10

“SonarQube bir projeyi taradığında tek bir sorun tipi üretmiyor; farklı kategorilerde değerlendirme yapıyor.

Bug, uygulamanın yanlış çalışmasına sebep olabilecek mantıksal hatalar. Örneğin JavaScript’te unreachable code buna güzel bir örnek.

Vulnerability, doğrudan güvenlik açığıdır. Python’da hardcoded password gibi.

Code Smell, daha çok kalite tarafıyla ilgilidir; gereksiz tekrarlar, karmaşık fonksiyonlar, okunabilirliği bozan yapılar.

Security Hotspot, ilginç bir kategori. Bu tam olarak açık değil; ama potansiyel güvenlik riski taşıyor. Manuel inceleme gerekir. Mesela eval kullanımı.

Bu kavramları bilmek önemli çünkü birazdan canlı demoda her birinin gerçek örneklerini göstereceğim.”

Slayt 11

“SonarQube, sadece hataları listelemiyor; aynı zamanda bunların ne kadar kritik olduğunu da belirliyor.

Blocker seviyesindeki bir hatayı gördüğünüz anda build’i durdurmak isteyiz. Critical seviyede ise güvenlik veya iş mantığı açısından ciddi bir etki vardır.

Major, Minor ve Info ise sırasıyla daha düşük etkili hatalar.

Bu sınıflandırma, Quality Gate kurallarının temelini oluşturuyor.”

Slayt 13 - 14

“Quality Gate, SonarQube’ün karar mekanizmasıdır. Yani taramadan sonra ‘Bu kod şu anki haliyle kabul edilebilir mi, yoksa build’i durdurmalı mıyım?’ sorusuna cevap verir.

Varsayılan bir Quality Gate geliyor ama çoğu zaman ekipler kendi kurallarını oluşturur. Mesela:

Yeni bug = 0

Yeni vulnerability = 0

Coverage ≥ %70

Bu şartlardan biri bile sağlanmazsa Quality Gate kırmızıya düşer ve pipeline durur. Birazdan GitHub Actions'da bunu canlı şekilde göstereceğim."

Slayt 15 - 16

"SonarQube'ün analiz sürecinin temelinde 'rules', yani kurallar bulunuyor. Bu kurallar dil özelinde binlerce farklı hatayı yakalayabiliyor. Güvenlik kuralları OWASP ve CWE gibi uluslararası standartlarla uyumlu; bu sayede gerçekten kritik güvenlik açıklarını yakalayabiliyor.

Bu kurallar, Quality Profile adını verdiğimiz yapılar içinde gruplanıyor. Bir proje hangi Quality Profile'a bağlıysa o profile içindeki kurallar aktif olur. Bu profilleri özelleştirebilir, kuralları açıp kapatabilir veya yeni kurallar ekleyebiliriz.

Her bir rule aslında bir mini politika: bir severity değeri var, hangi kategoriye girdiği belli ve bir de remediation cost dediğimiz 'düzeltme süresi' tahmini bulunuyor.

Ekranda birkaç örnek görebilirsiniz: Python'daki os.system çağrısı komut enjeksiyonu riskine karşı bir vulnerability rule'u tetikliyor. JavaScript'teki SQL string birleştirme SQL Injection rule'unu tetikliyor. Hardcoded secret, güvenlik açısından doğrudan bir vulnerability olarak işaretleniyor.

Bir başka önemli nokta da şu: SonarLint, SonarQube ile aynı rule set'i kullanıyor. Yani bu kurallar sadece pipeline'da değil, geliştiricinin IDE'sinde de aynı şekilde çalışıyor. Bu sayede 'Shift Left Security' yaklaşımını pratik olarak uygulamış oluyoruz."

Slayt 17

"Bir issue bulunduğunda geliştirme ekipleri bununla ne yapacağına karar vermeli. SonarQube bu konuda güçlü bir workflow sunuyor.

Issue'yu bir kullanıcıya atayabiliyoruz. Eğer SonarQube'ün yaptığı tespit yanlışsa False Positive diyebiliyoruz.

'Bu hatayı biliyoruz ama çözmeyeceğiz' gibi durumlar için Won't Fix mevcut.

Ayrıca Security Hotspot kategorisi, güvenlik açısından kritik noktalara göz atmayı gerektiriyor ama her hotspot doğrudan açık değildir.

Bu işlemleri de canlı demoda göstereceğim."

Slayt 18

"Artık SonarQube'ün ne yaptığını biliyoruz; şimdi sorumuz şu: Bu aracı DevOps sürecinin neresine ve nasıl bağlayacağız?

Aslında üç farklı entegrasyon modeli var:

Birinci yol, benim bu sunumda da kullandığım yöntem: VCS tarafında bir Application / OAuth entegrasyonu. Örneğin GitHub App veya GitLab Application tanımlayıp SonarQube'ü doğrudan repository yönetim sistemiyle konuşturabiliyoruz. SonarQube projeyi kendisi görüyor, webhook'larla tetiklenebiliyor.

İkinci yol, daha klasik olanı: CI pipeline içinde SonarScanner çalıştırmak. Burada repoda bir sonar-project.properties dosyamız oluyor, CI YAML içinde de Sonar token'ı kullanarak SonarQube'e sonucu gönderiyoruz.

Üçüncü yol ise tamamen lokal: Geliştirici kendi makinesine SonarScanner kurup komut satırından analiz çalıştırıyor. Bu yöntem özellikle PoC, offline test ortamları veya CI'ye entegre etmeden önce deneme yapmak için kullanışlı.

Bir sonraki slaytlarda özellikle ikinci ve üçüncü yöntemi kod örnekleriyle göstereceğim; ama bu sunumda canlı demoda birinci yolu yani GitHub App entegrasyonunu kullanacağım."

Slayt 19

"İkinci entegrasyon yaklaşımı, klasik CI pipeline + SonarScanner modeli.

Burada repomuzun köküne bir sonar-project.properties dosyası koyuyoruz. Bu dosya, SonarQube'e 'Bu proje hangi dilde, kaynak dosyalar nerede, testler nerede?' gibi bilgileri veriyor.

Ardından CI tarafında – burada örnek olarak GitHub Actions verdim – bir job tanımlıyoruz. Önce kodu checkout ediyoruz, gerekiyorsa runtime ortamını hazırlıyoruz, sonra da SonarScanner action'ını çağırıyoruz.

Dikkat ederseniz, SonarQube'e bağlantı için iki kritik değişken kullanıyorum: SONAR_HOST_URL ve SONAR_TOKEN.

SONAR_TOKEN, SonarQube'de oluşturduğumuz kişisel erişim token'ı; GitHub tarafında secret olarak saklıyoruz. Bu modelin avantajı şu: Her push işleminde pipeline otomatik analiz yapıyor ve isterseniz Quality Gate sonucuna göre build'i kırabiliyoruz.

Ben kendi demosunda GitHub App kullandım; ama bu yöntem de özellikle daha esnek CI senaryoları için oldukça yaygın."

Slayt 21

"Üçüncü entegrasyon modeli ise tamamen lokal.

Geliştirici kendi makinesine SonarScanner'ı kuruyor, proje kökünde bir sonar-project.properties dosyası varsa sadece sonar-scanner komutunu çalıştırması yetiyor.

Komut örneğinde gördüğünüz gibi, sonar.host.url ile SonarQube sunucusunun adresini, sonar.login ile de token bilgisini veriyoruz.

Bu yöntem özellikle CI pipeline'ı olmayan eski projelerde, Proof of Concept yapmak istediğimiz durumlarda veya offline araştırmalarda çok kullanışlı.

Güzel tarafı şu: CI olsun ya da olmasın, SonarQube tarafında aynı kurallar, aynı Quality Gate ve aynı dashboard kullanılıyor. Sadece tetikleme noktası değişiyor."

Slayt 22

"SonarQube sadece projenin içinde analiz yapmakla kalmıyor; kalite durumunu dışarıya da yansıtabilen bir badge mekanizması sunuyor.

Ekranda gördüğünüz gibi Quality Gate, Bugs, Vulnerabilities, Code Smells gibi her metrik için birer badge üretebiliyoruz. Bunları GitHub'daki README dosyamıza koyduğumuzda, proje durumunu anlık olarak dışarıya açmış oluyoruz.

Bu özellikle ekiplerde standartlaştırma için önemli. Mesela bir proje Quality Gate'den geçmiyorsa, badge otomatik olarak kırmızıya dönüyor ve bunu repoya bakan herkes direkt görebiliyor.

Badge aslında görünüş olarak küçük bir ikon, ama DevSecOps tarafında çok büyük bir kültür etkisi var: kalitenin görünürlüğünü artırıyor.

Birazdan canlı demoda test repomun badge'inin ne durumda olduğunu da göstereceğim — muhtemelen kırmızı çünkü içinde bilerek yerleştirdiğim hatalar var 😃"

Slayt 23 - 24 - 25

"SonarQube üç ana sürümle geliyor: Community, SonarCloud ve Enterprise.

Community Edition, açık kaynak ve tamamen ücretsiz. On-prem kurulum yapabiliyoruz ve temel SAST, Quality Gate, Issue yönetimi gibi çekirdek özelliklerin hepsi mevcut. Ben bu sunumda Community Edition kullanıyorum. Kişisel projeler veya küçük takımlar için fazlasıyla yeterli.

SonarCloud, SaaS modeliyle çalışan sürüm. Kurulum gerektirmiyor; özellikle GitHub, GitLab gibi platformlarla doğrudan entegre çalışıyor. Public repolar ücretsiz, private repolar ise ücretli. CI YAML yazmadan repository push edince otomatik tetiklenmesi büyük kolaylık sağlıyor.

Enterprise Edition ise büyük organizasyonlara yönelik. Portfolio yönetimi, kurumsal raporlama, gelişmiş güvenlik kuralları, OWASP/CWE tabanlı derin analizler, branch/PR analysis gibi yetenekler burada geliyor. Ayrıca yüksek ölçekli yapılarda distributed analiz imkanları sunuyor.

Özetle:

Community → bireysel veya küçük ekip

SonarCloud → kurulum istemeyen modern GitHub ekipleri

Enterprise → büyük şirketler, güvenlik standartları ve governance gerektiren ortamlar

Biz bugün Community sürüm üzerinde canlı bir demo yapacağız."

DEMO

SON