ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΩΣ

ΣΧΟΛΗ ΤΕΧΝΟΛΟΓΙΩΝ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΕΠΙΚΟΙΝΩΝΙΩΝ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

Πτυχιακή Εργασία

Τίτλος Πτυχιακής Εργασίας	Greek «Αυτοματοποιημένος Έλεγχος Ασφάλειας Εφαρμογές Web μέσω διεργασιών του GitHub» English «Automated Security Auditing for Web Applications through GitHub Pipelines»
Ονοματεπώνυμο Φοιτητή	Παναγιώτης Χονδρόπουλος
Πατρώνυμο	Γεώργιος
Αριθμός Μητρώου	Π16156
Επιβλέπων	Κωνσταντίνος Πατσάκης, Καθηγητής

Ιούνιος 2026

Copyright ©

Περίληψη

Η εργασία παρουσιάζει την ανάπτυξη μιας αυτοματοποιημένης ροής ελέγχου ασφάλειας web εφαρμογών μέσω GitHub Actions. Συνδυάζει εργαλεία στατικής και δυναμικής ανάλυσης για τον εντοπισμό ευπαθειών, εκτεθειμένων μυστικών, προσωπικών δεδομένων και ευάλωτων εξαρτήσεων. Τα αποτελέσματα συγκεντρώνονται και αναλύονται από Agent Τεχνητής Νοημοσύνης, ο οποίος παράγει μια κατανοητή αναφορά με προτεινόμενες ενέργειες διόρθωσης.

Λέξεις-κλειδιά:

GitHub Actions, Ασφάλεια Web Εφαρμογών, Αυτοματοποιημένος Έλεγχος Ασφάλειας, SAST, DAST, Τεχνητή Νοημοσύνη

Abstract

This thesis presents an automated security auditing pipeline for web applications using GitHub Actions. It combines static and dynamic analysis tools to detect vulnerabilities, exposed secrets, personal data, and vulnerable dependencies. The results are collected and analyzed by an AI Agent, which produces a clear security report with practical remediation recommendations.

Key Words:

GitHub Actions, Web Application Security, Automated Security Auditing, SAST, DAST, Artificial Intelligence

Κεφάλαιο 1: Εισαγωγή

Στη σύγχρονη ανάπτυξη λογισμικού, η ασφάλεια και η προστασία της ιδιωτικότητας δεν μπορούν να αποτελούν απλώς το τελευταίο βήμα πριν την κυκλοφορία μιας εφαρμογής. Η ανάγκη για έγκαιρο εντοπισμό ευπαθειών έχει οδηγήσει στη χρήση πολλών εργαλείων ελέγχου. Αυτό δημιουργεί ένα νέο πρακτικό πρόβλημα: τα εργαλεία αυτά παράγουν τεράστιο όγκο ακατέργαστων δεδομένων, σε μορφές όπως JSON, SARIF ή απλά αρχεία καταγραφής. Ως αποτέλεσμα, η επίλυση των σφαλμάτων γίνεται μια χρονοβόρα και δύσκολη διαδικασία.

Η εργασία επιχειρεί να αντιμετωπίσει αυτήν ακριβώς την πρόκληση. Σκοπός της είναι ο σχεδιασμός και η υλοποίηση μιας αυτοματοποιημένης ροής ελέγχων, η οποία αρχικά βοηθά στην ανίχνευση ευπαθειών στον κώδικα, και στη συνέχεια στην επεξήγηση των ευπαθειών πριν αυτές γίνουν επικίνδυνες για το σύστημα.

Για την επίτευξη αυτού του στόχου, η εργασία ενσωματώνει παράλληλους στατικούς ελέγχους με τη βοήθεια των εργαλείων Gitleaks, CodeQL, Trivy και Custom PII Scanner, καθώς και δυναμικούς ελέγχους σε μια ενεργή εφαρμογή με τη βοήθεια των εργαλείων OWASP ZAP και SQLMap.

Στη συνέχεια προχωράει στη διαχείριση των αποτελεσμάτων, συγκεντρώνοντας τα δεδομένα ευρημάτων των ελέγχων και τροφοδοτώντας τα στη συνέχεια σε έναν Agent Τεχνητής Νοημοσύνης. Ο Agent λειτουργεί ως ενεργός σύμβουλος ασφαλείας, αναλύοντας τα ευρήματα και επιστρέφοντας μια απλή, πρακτική αναφορά σε φυσική γλώσσα, η οποία εξηγεί τον κίνδυνο και προτείνει συγκεκριμένα βήματα διόρθωσης.

Κεφάλαιο 2: Εργαλεία και Κριτήρια Επιλογής

Η επιλογή των εργαλείων για το αυτοματοποιημένο security pipeline έγινε με ορισμένα βασικά κριτήρια, τόσο μεμονωμένα όσο και ως σύνολο. Κάθε εργαλείο πρέπει να καλύπτει έναν συγκεκριμένο τύπο κινδύνου, έτσι ώστε να δημιουργηθεί μια λογική αλυσίδα ελέγχων, η οποία να μπορεί να εντοπίζει τα προβλήματα νωρίς, πριν ο κώδικας ενσωματωθεί στο κύριο branch του repository.
Τα βασικά κριτήρια επιλογής των εργαλείων ήταν πέντε:

1. **Αυτοματοποίηση μέσω GitHub Actions:** Κάθε εργαλείο έπρεπε να μπορεί να εκτελεστεί αυτόματα, χωρίς χειροκίνητη παρέμβαση.
2. **Τεχνική κάλυψη:** Κάθε εργαλείο έπρεπε να προσθέτει μια ξεχωριστή μορφή ελέγχου, χωρίς να καλύπτει τη λειτουργία κάποιου άλλου. Γι' αυτό, τα εργαλεία που επιλέχθηκαν εξειδικεύονται σε διαφορετικούς τομείς ασφάλειας και ιδιωτικότητας.

3. **Τρόπος παρουσίασης των αποτελεσμάτων:** Ο κύριος στόχος ενός εργαλείου ασφάλειας είναι να εντοπίζει προβλήματα, όμως είναι εξίσου σημαντικό τα αποτελέσματα που παράγει να μπορούν να διαβαστούν και να αξιοποιηθούν με ευκολία. Για τον λόγο αυτό δόθηκε μεγάλη σημασία στην παραγωγή αναφορών όπως αρχεία JSON, HTML, SARIF και GitHub Actions artifacts.
4. **Μηδενικό κόστος χρήσης:** Για τις ανάγκες της εργασίας επιλέχθηκαν εργαλεία ελεύθερου κώδικα (open-source) ή διαθέσιμα δωρεάν στο οικοσύστημα του GitHub.
5. **Απλότητα:** Το project δεν είχε στόχο να αναπαραγάγει ένα υπερβολικά πολύπλοκο περιβάλλον, αλλά να δείξει με καθαρό τρόπο πώς μπορεί να στηθεί ένα λειτουργικό και επεκτάσιμο σύστημα ασφάλειας. Έτσι, αποφεύχθηκαν εργαλεία που θα απαιτούσαν πολύπλοκη υποδομή.

Το project βασίστηκε στη λογική της σταδιακής κάλυψης διαφορετικών επιπέδων κινδύνου. Για τον λόγο αυτό, τα εργαλεία χωρίστηκαν σε δύο διαφορετικές κατηγορίες:

Η πρώτη κατηγορία αφορά τη στατική ανάλυση, δηλαδή τον έλεγχο του κώδικα και των αρχείων του repository χωρίς να χρειάζεται να εκτελεστεί η εφαρμογή. Σε αυτή την κατηγορία εντάσσονται εργαλεία όπως το Gitleaks, το CodeQL, το Trivy και το custom PII scanner.

Η δεύτερη κατηγορία αφορά τη δυναμική ανάλυση, όπου η εφαρμογή εκτελείται και εξετάζεται σε πραγματικό χρόνο. Σε αυτή την κατηγορία χρησιμοποιήθηκαν το OWASP ZAP και το SQLMap.

Στο τέλος των δύο αναλύσεων προστέθηκε ένας LLM Agent, ο οποίος λειτουργεί ως ένα επιπλέον επίπεδο ανάλυσης και επεξήγησης των αποτελεσμάτων.

Η συνολική στρατηγική ήταν να επιλεγούν εργαλεία που συνεργάζονται αρμονικά μεταξύ τους, ενώ ταυτόχρονα καλύπτουν διαφορετικού είδους ελέγχους:

- Το Gitleaks ελέγχει αν υπάρχουν εκτεθειμένα μυστικά (secrets).
- Το CodeQL εξετάζει τη λογική του κώδικα για πιθανά κενά ασφαλείας (vulnerabilities).
- Το Trivy ελέγχει εξαρτήσεις και κινδύνους στην αλυσίδα εφοδιασμού (supply-chain).
- Ο PII scanner εντοπίζει προσωπικά δεδομένα που δεν πρέπει να υπάρχουν στο repository.
- Το OWASP ZAP εξετάζει την εφαρμογή ενώ αυτή τρέχει, εντοπίζοντας προβλήματα που φαίνονται μέσα από τα HTTP αιτήματα και τις αποκρίσεις.
- Το SQLMap εστιάζει πιο συγκεκριμένα στον εντοπισμό επιθέσεων SQL injection.
- Ο LLM Agent συνδυάζει τα αποτελέσματα, τα απλοποιεί και τα μετατρέπει σε πιο κατανοητές οδηγίες διόρθωσης.

Με αυτόν τον τρόπο κάθε εργαλείο της εργασίας έχει τον δικό του ρόλο.

2.1 Στατική Ανάλυση: Επιλογές Εργαλείων και Αιτιολόγηση

Η στατική ανάλυση αποτελεί το πρώτο workflow του συστήματος. Ως σύνολο έχει συγκεκριμένο ρόλο και πλεονεκτήματα, καθώς εκτελείται γρήγορα, δεν απαιτεί εκτέλεση της εφαρμογής και εφαρμόζεται απευθείας στον πηγαίο κώδικα. Αυτά τα χαρακτηριστικά την καθιστούν τον κατάλληλο μηχανισμό ελέγχου σε επίπεδο Pull Requests και Git pushes, ώστε τα προβλήματα να εντοπίζονται πριν καν ενσωματωθούν στον βασικό κώδικα.

Για το project, η στατική ανάλυση αντιμετωπίστηκε ως ένα ολοκληρωμένο σύστημα διαχείρισης συγκεκριμένων κατηγοριών κινδύνου. Οι κατηγορίες αυτές επιλέχθηκαν με βάση τις πιο κρίσιμες και ρεαλιστικές απειλές που αντιμετωπίζει μια εφαρμογή:

- **Διαρροή Μυστικών:** Αφορά ευαίσθητα δεδομένα ταυτοποίησης που γίνονται commit κατά λάθος μέσα στον κώδικα. Μιλάμε κυρίως για API keys, tokens και passwords. Επειδή τέτοιου είδους διαρροές είναι πολύ σημαντικές και μπορούμε να τις εντοπίσουμε με μεγάλη σιγουριά, είναι λογικό το σύστημα να κάνει "fail" αμέσως αν βρει κάτι τέτοιο.
- **Ευπάθειες Λογικής Κώδικα:** Αφορά ανασφαλή μοτίβα προγραμματισμού μέσα στον ίδιο τον πηγαίο κώδικα. Τέτοια παραδείγματα είναι ο κίνδυνος για SQL injection ή command injection. Ο στόχος είναι να βρούμε εκμεταλλεύσιμα bugs αναλύοντας το πώς ρέουν τα δεδομένα (data flow) στην εφαρμογή.
- **Ευάλωτες Εξαρτήσεις:** Αφορά ευπάθειες που προέρχονται από τα dependencies. Χρησιμοποιώντας third-party βιβλιοθήκες, τυχόν γνωστές ευπάθειες σε αυτές σημαίνουν ότι το project κληρονομεί αυτόματα τον κίνδυνο. Ο έλεγχος αρχείων όπως το requirements.txt για ευάλωτες εκδόσεις μας επιτρέπει να βρούμε γνωστές ευπάθειες σε όλο το project με ελάχιστο κόπο.
- **Έκθεση Προσωπικών Δεδομένων:** Αποτελεί ρίσκο ιδιωτικότητας. Πολλές φορές μένουν ξεχασμένα στον κώδικα ευαίσθητα δεδομένα που μπορούν να ταυτοποιήσουν ένα πρόσωπο, όπως τηλέφωνα, emails ή IBAN.

Για την αντιμετώπιση του κάθε προβλήματος ξεχωριστά, αξιολογήθηκαν διάφορες λύσεις. Για κάθε κατηγορία κινδύνου, επιλέχθηκε τελικά ένα κατάλληλο εργαλείο το οποίο καλύπτει τα κριτήρια του project.

Το Gitleaks επιλέχθηκε για τον έλεγχο εκτεθειμένων μυστικών, όπως API keys, tokens, passwords και άλλα credentials. Το βασικό πλεονέκτημα του Gitleaks είναι ότι ειδικεύεται σε αυτόν τον τύπο ελέγχου. Σαρώνει το repository, καθώς και το ιστορικό του Git, και εντοπίζει μοτίβα που μοιάζουν με credentials. Πριν επιλεγεί το Gitleaks, εξετάστηκαν και άλλες επιλογές. Μία εναλλακτική ήταν το TruffleHog, το οποίο επίσης χρησιμοποιείται για τον ίδιο τύπου έλεγχο. Όμως, για τις ανάγκες του συγκεκριμένου project θεωρήθηκε πιο σύνθετο από όσο χρειαζόταν, καθώς παράγει περισσότερα ευρήματα που απαιτούν πολύπλοκη διαχείριση. Για το project, το Gitleaks ενσωματώνεται εύκολα στο GitHub Actions workflow και μπορεί να ρυθμιστεί ώστε να αποτυγχάνει το pipeline όταν βρίσκει κάτι, διατηρώντας την πολυπλοκότητα χαμηλή. Έτσι, επιλέχθηκε ως το εργαλείο εύρεσης μυστικών.

Το CodeQL επιλέχθηκε για τον Code-level έλεγχο, εντοπίζοντας ευπάθειες στη λογική του κώδικα, όπως κίνδυνο για injections και ανασφαλές data flow. Το βασικό πλεονέκτημα του

CodeQL είναι ότι αναλύει τον κώδικα με βάθος, καθώς χτίζει ένα μοντέλο του κώδικα και παρακολουθεί τη ροή των δεδομένων εφαρμόζοντας κανόνες ασφαλείας, ώστε να εντοπίζει πραγματικά bugs. Πριν επιλεγεί το CodeQL, εξετάστηκαν και άλλες επιλογές. Μία εναλλακτική ήταν το Bandit, το οποίο επίσης χρησιμοποιείται για τον ίδιο τύπο ελέγχου. Όμως, για τις ανάγκες του συγκεκριμένου project απορρίφθηκε καθώς είναι αυστηρά προσανατολισμένο μόνο στην Python. Για το project, το CodeQL ενσωματώνεται εύκολα στο οικοσύστημα του GitHub και τα Actions, ενώ τα ευρήματά του εμφανίζονται απευθείας στο Security tab, διατηρώντας την πολυπλοκότητα χαμηλή αλλά τη μορφή παρουσίασης απλή και κατανοητή. Έτσι, επιλέχθηκε ως το εργαλείο εντοπισμού ευπαθειών λογικής κώδικα.

Το Trivy επιλέχθηκε για την ανάλυση λογισμικού (SCA), ελέγχοντας τις εξωτερικές βιβλιοθήκες για γνωστές ευπάθειες και κινδύνους στην εφοδιαστική αλυσίδα. Το βασικό πλεονέκτημα του Trivy είναι ότι προσφέρει τεράστια κάλυψη με ελάχιστη προσπάθεια. Σαρώνει τα αρχεία κειμένου που δηλώνουμε ποιες εξωτερικές βιβλιοθήκες χρησιμοποιεί η εφαρμογή και συγκρίνει τις εκδόσεις τους με γνωστές βάσεις δεδομένων ευπαθειών. Πριν επιλεγεί το Trivy, εξετάστηκαν και άλλες επιλογές. Μία εναλλακτική ήταν το OSV-Scanner, το οποίο όμως προσφέρει πιο περιορισμένο εύρος ανάλυσης. Μια δεύτερη επιλογή ήταν το Snyk, το οποίο παρότι είναι ισχυρό, απαιτεί δημιουργία λογαριασμού και εξάρτηση από εμπορική πλατφόρμα, κάτι που δεν ταίριαζε στα κριτήρια της εργασίας. Για το project, το Trivy ενσωματώνεται εύκολα στο GitHub Actions workflow, τρέχει εντελώς τοπικά και δωρεάν, και μπορεί να ρυθμιστεί ως προς το με τι είδους ευρήματα θα αποτυγχάνει το pipeline, διατηρώντας την πολυπλοκότητα χαμηλή. Έτσι, επιλέχθηκε ως το εργαλείο ανάλυσης εξαρτήσεων.

Ο custom PII scanner επιλέχθηκε για τον έλεγχο πιθανών διαρροών προσωπικών δεδομένων, όπως τηλέφωνα, emails και IBAN. Ο custom scanner, ο οποίος γράφτηκε σε Python, σαρώνει τα αρχεία χρησιμοποιώντας Regular Expressions για αναγνώριση μοτίβων. Για το project, ο custom PII scanner ενσωματώνεται εύκολα στο GitHub Actions workflow ως ένα αυτόνομο script που εκτελείται κατά το build, διατηρώντας την πολυπλοκότητα και τις εξαρτήσεις από εξωτερικές υπηρεσίες πολύ χαμηλά. Έτσι, επιλέχθηκε ως το ιδανικό εργαλείο για την ανάδειξη του privacy scanning.

Τα τέσσερα αυτά εργαλεία λειτουργούν συμπληρωματικά, καλύπτοντας τέσσερις διαφορετικούς άξονες κινδύνου. Το Gitleaks διασφαλίζει ότι δεν διαρρέουν ευαίσθητα credentials, το CodeQL ελέγχει τη λογική του ίδιου του πηγαίου κώδικα για εκμεταλλεύσιμα bugs, το Trivy αποτρέπει την εισαγωγή ευπαθειών μέσω τρίτων βιβλιοθηκών, και ο PII scanner προστατεύει τα προσωπικά δεδομένα των χρηστών. Συνδυάζοντας αυτές τις λύσεις, το σύστημα καλύπτει και τα τέσσερα είδη κινδύνου, εξασφαλίζοντας ότι το project ελέγχεται αυτοματοποιημένα και γρήγορα πριν προχωρήσει στα επόμενα στάδια ανάπτυξης.

2.2 Δυναμική Ανάλυση: Επιλογές Εργαλείων και Αιτιολόγηση

Η στατική ανάλυση είναι το πρώτο και βασικό βήμα, αλλά δεν μπορεί να καλύψει από μόνη της όλους τους κινδύνους. Ο λόγος είναι ότι πολλά κενά ασφαλείας δεν φαίνονται απλά διαβάζοντας τον κώδικα στο repository, αλλά εμφανίζονται μόνο όταν η εφαρμογή "τρέχει". Για τον λόγο αυτό, στο project προστέθηκε το workflow της δυναμικής ανάλυσης.

Η βασική διαφορά ανάμεσα στα δύο είδη ανάλυσης είναι ότι στο δυναμικό σκανάρισμα (DAST) ο έλεγχος γίνεται με μια demo εφαρμογή να "τρέχει" κανονικά στο background. Πιο συγκεκριμένα, στέλνοντας διάφορα αιτήματα και παρατηρώντας τις απαντήσεις που επιστρέφει το σύστημα, προσπαθούμε να εντοπίσουμε ευπάθειες και κινδύνους που μπορεί να έχει η εφαρμογή. Για να γίνει αυτό με ασφάλεια, το σύστημα αντιγράφει αυτόματα αυτή τη demo εφαρμογή και την “τρέχει” τοπικά μέσα στο GitHub Actions runner.

Με αυτή την προσέγγιση, το κομμάτι δυναμικής ανάλυσης του project στοχεύει σε δύο βασικές κατηγορίες κινδύνου:

- **Runtime Web Ευπάθειες:** Πρόκειται για προβλήματα συμπεριφοράς στο web επίπεδο. Εδώ ανήκουν κακές ρυθμίσεις ή κακό configuration που γίνονται αντιληπτά μόνο όταν η εφαρμογή απαντάει σε πραγματικά αιτήματα, όπως ελλιπή security headers, κακή διαχείριση sessions/cookies και προβλήματα στο routing.
- **Ενεργές Επιθέσεις σε Endpoints:** Πρόκειται για ενεργή δοκιμή στοχευμένων σημείων (όπως URL parameters). Ο στόχος είναι να ελεγχθεί αν δεδομένα από τον χρήστη περνούν ανεξέλεγκτα σε ερωτήματα προς τη βάση δεδομένων, επιτρέποντας ενδεχομένως αλλαγή ή διαγραφή δεδομένων.

Για την αντιμετώπιση αυτών των δύο σεναρίων κατά την εκτέλεση της εφαρμογής, επιλέχθηκαν δύο συγκεκριμένα εργαλεία τα οποία εκτελούνται πάνω στην τοπική demo εφαρμογή.

Το OWASP ZAP επιλέχθηκε ως το βασικό εργαλείο για τον γενικό έλεγχο των web ευπαθειών. Το μεγάλο του πλεονέκτημα είναι ότι σαρώνει συνολικά τη συμπεριφορά της εφαρμογής στο web επίπεδο, αναζητώντας αδυναμίες στα HTTP responses. Κατά τη ρύθμισή του, υπήρχε η επιλογή ανάμεσα σε baseline scan (γρήγορο και ιδανικό για συνεχή χρήση σε συχνά pull requests) και full scan (πιο λεπτομερές, αλλά και πιο αργό). Για τις ανάγκες του project επιλέχθηκε το full scan. Παρόλο που σε ένα production περιβάλλον θα ήταν πιο πολύπλοκο και αργό, στο συγκεκριμένο project τρέχει απέναντι σε μια μικρή demo εφαρμογή, οπότε μας δίνει την ευκαιρία να δείξουμε τις πραγματικές δυνατότητες ενός ολοκληρωμένου δυναμικού ελέγχου, χωρίς να μας επηρεάζει ο χρόνος εκτέλεσης. Έτσι, το ZAP επιλέχθηκε ως ο γενικός δυναμικός σαρωτής.

Το SQLMap επιλέχθηκε ως ένα εξειδικευμένο εργαλείο αποκλειστικά για SQL injection testing, καθώς εστιάζει σε έναν πολύ συγκεκριμένο και κρίσιμο τύπο επίθεσης. Χρειάζεται έναν ακριβή στόχο, δηλαδή ένα συγκεκριμένο endpoint (π.χ. ένα URL με παραμέτρους), και αρχίζει να στέλνει διάφορες τιμές για να παρατηρήσει αν μπορεί να χειραγωγήσει τα SQL queries στο background. Αν και γενικά εργαλεία όπως το ZAP βρίσκουν τέτοια θέματα, το SQLMap είναι πολύ πιο ισχυρό στη συγκεκριμένη κατηγορία.

Τα δύο αυτά εργαλεία λειτουργούν συμπληρωματικά, καλύπτοντας διαφορετικούς άξονες της δυναμικής ανάλυσης. Το ZAP ελέγχει συνολικά τη web συμπεριφορά της εφαρμογής, ενώ το SQLMap εστιάζει σε συγκεκριμένα endpoints για αδυναμίες σε βάσεις δεδομένων. Μαζί, δημιουργούν reports που ενσωματώνονται στο υπόλοιπο σύστημα.

2.3 Ο Ρόλος του Agent Τεχνητής Νοημοσύνης

Οι στατικοί και δυναμικοί έλεγχοι του project κάνουν καλή δουλειά στον εντοπισμό διαφορετικών κατηγοριών κινδύνου, όμως τα αποτελέσματα που παράγουν είναι συχνά μεγάλα, τεχνικά και δύσκολα στην άμεση αξιοποίησή τους. Κάθε εργαλείο παράγει διαφορετικού τύπου αρχεία (JSON, HTML, terminal logs), τα οποία είναι τεράστια και γεμάτα χαώδεις τεχνικές λεπτομέρειες. Έτσι, ενώ το σύστημα βρίσκει τα προβλήματα, είναι σημαντική η προσθήκη κάποιου εργαλείου οργάνωσης, επεξήγησης και απλοποίησης των αποτελεσμάτων.

Για αυτόν τον λόγο, προστέθηκε ένας LLM Agent, ο οποίος αναλαμβάνει ουσιαστικά την επεξήγηση των τεχνικών προβλημάτων, έτσι ώστε η έξοδος να είναι πιο συγκεκριμένη και κατανοητή. Ο Agent είναι τοποθετημένος στο τέλος του pipeline δυναμικού ελέγχου και ο ρόλος του είναι να πάρει όλα τα reports, να τα ενώσει, να βάλει σε προτεραιότητα τα πιο κρίσιμα ευρήματα και να τα εξηγήσει με απλά λόγια, χωρίς να μένει μόνο στην αναφορά του προβλήματος, αλλά προτείνοντας και λύσεις.

Το Gemini 3.1 Flash-Lite επιλέχθηκε ως το μοντέλο τεχνητής νοημοσύνης για την υλοποίηση του LLM Agent. Το βασικό πλεονέκτημα του συγκεκριμένου μοντέλου είναι ότι είναι εξαιρετικά ελαφρύ και γρήγορο. Αυτά είναι χαρακτηριστικά που ταιριάζουν στον ρόλο του Agent στο project, καθώς και στο μικρό μέγεθος της εφαρμογής. Αυτό του επιτρέπει να διαβάζει τα reports των scanners και να παράγει τις συνόψεις και τις προτάσεις του πολύ γρήγορα, χωρίς να καθυστερεί τη συνολική εκτέλεση της διαδικασίας.

Κεφάλαιο 3: Αρχιτεκτονική και Υλοποίηση του Συστήματος Ασφαλείας

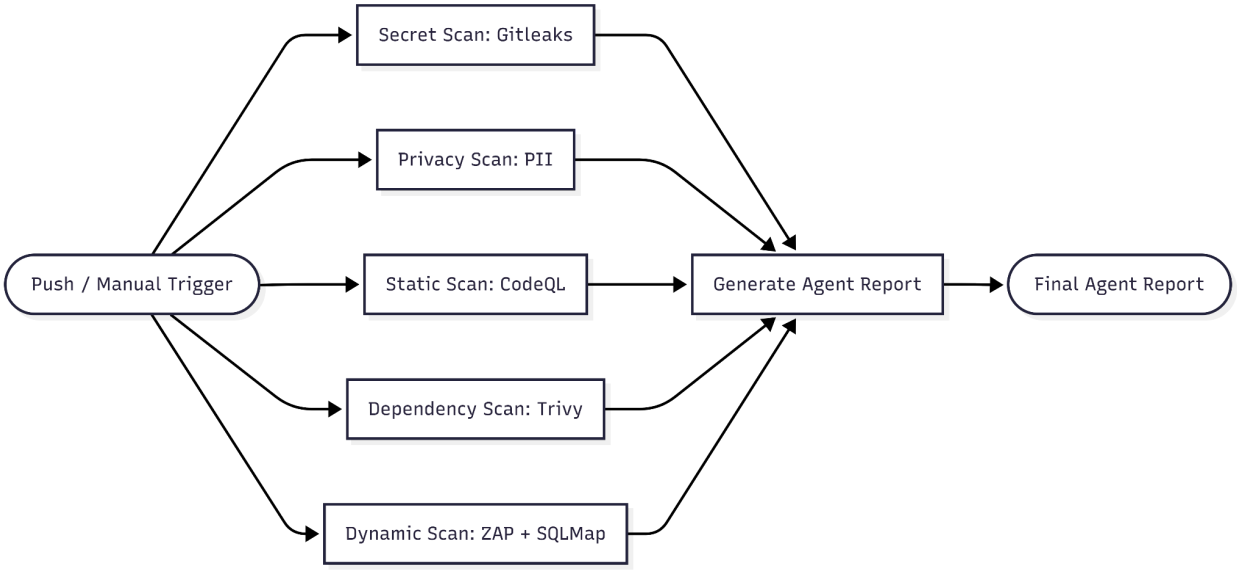
3.1 Αρχιτεκτονική και Λογική του συστήματος

Ο βασικός σκοπός του project είναι η δημιουργία ενός μηχανισμού, ο οποίος ελέγχει τον κώδικα για διάφορα κενά ασφαλείας, πριν αυτός φτάσει στο τελικό στάδιο παραγωγής. Η υλοποίηση του μηχανισμού βασίζεται στα GitHub Actions, επειδή επιτρέπει σε όλους τους ελέγχους να εκτελούνται αυτόματα, απευθείας μέσα στο περιβάλλον του repository.

Το σύστημα, χωρίζεται λειτουργικά σε τρία βασικά επίπεδα:

1. **Το επίπεδο αυτοματοποίησης:** Μέσω του GitHub Actions, αναλαμβάνει να ξεκινήσει τη διαδικασία, να διαχειριστεί τα παράλληλα jobs και να μεταφέρει τα δεδομένα (artifacts) από το ένα στάδιο στο επόμενο.
2. **Το επίπεδο των ελέγχων:** Περιλαμβάνει τα εργαλεία στατικής και δυναμικής ανάλυσης, τα οποία ελέγχουν τον κώδικα και την εφαρμογή κατά την εκτέλεσή της.
3. **Το επίπεδο παραγωγής της αναφοράς του Agent:** Το τελικό στάδιο, όπου όλα τα αποτελέσματα των ελέγχων μαζεύονται και μεταφράζονται από έναν AI agent σε μια κατανοητή και πρακτική αναφορά.

Για να συνδεθούν αυτά τα τρία επίπεδα αποδοτικά, η σχεδίαση του συστήματος ακολουθεί την εξής λογική. Μόλις ενεργοποιηθεί το pipeline, ξεκινούν να εκτελούνται παράλληλα και ανεξάρτητα πολλαπλοί έλεγχοι: το secret scan, το PII scan, το SAST scan, το dependency scan και το dynamic scan. Κάθε έλεγχος γίνεται ταυτόχρονα σε δικό του απομονωμένο περιβάλλον (runner). Μόλις ολοκληρωθούν όλοι οι έλεγχοι, τα παραγόμενα reports συγκεντρώνονται σε ένα κεντρικό σημείο για τη δημιουργία του τελικού AI report.



Η συγκεκριμένη αρχιτεκτονική επιλέχθηκε επειδή προσφέρει δύο βασικά πλεονεκτήματα:

- **Ταχύτητα εκτέλεσης:** Κάθε έλεγχος λειτουργεί εντελώς αυτόνομα και παράλληλα, έτσι ο χρόνος που απαιτείται για να ολοκληρωθεί το σύστημα μειώνεται.
- **Ανθεκτικότητα στα σφάλματα:** Αν ένα μεμονωμένο εργαλείο αποτύχει (είτε λόγω τεχνικού προβλήματος, είτε επειδή εντόπισε κάποιο βασικό κίνδυνο), η διαδικασία συνεχίζει κανονικά, συγκεντρώνει τα αποτελέσματα από τα υπόλοιπα επιτυχημένα εργαλεία και παραδίδει στον developer την αναφορά.

3.2 Σχεδίαση του GitHub Actions Workflow και Triggers

Ο συντονισμός ολόκληρου του project γίνεται μέσω του αρχείου full-security-scan.yml. Το συγκεκριμένο αρχείο YAML λειτουργεί ως το κέντρο του συστήματος, περιγράφοντας πότε θα ξεκινήσουν οι έλεγχο (trigger), ποια βήματα θα ακολουθηθούν (jobs) και πώς θα διαχειριστούν τα παραγόμενα αρχεία (artifacts).

Η εκκίνηση των ελέγχων ορίζεται από δύο διαφορετικά triggers:

```
on:
  push:
    branches: ["main"]
  workflow_dispatch:
```

- Το push στο main branch εξασφαλίζει ότι οποιαδήποτε αλλαγή ή προσθήκη κώδικα στον κεντρικό κορμό της εφαρμογής θα ελεγχθεί αυτόματα, διαβεβαιώνοντας ότι κάθε νέα προσθήκη πληροί τις βασικές προϋποθέσεις ασφάλειας του project
- Το workflow_dispatch δίνει στον developer την ικανότητα να εκκινεί το σύστημα ελέγχου χειροκίνητα μέσα από το περιβάλλον του GitHub, κάτι που είναι ιδιαίτερα χρήσιμο όταν θέλει να επιβεβαιώσει την κατάσταση του συστήματος χωρίς να έχει προηγηθεί κάποιο commit.

Το αρχείο εστιάζει στον διαχωρισμό των λειτουργιών σε ανεξάρτητα jobs. Τα πέντε πρώτα jobs (secret_scan, pii_scan, sast_scan, dependency_scan, dynamic_scan) είναι σχεδιασμένα ώστε να μην έχουν καμία εξάρτηση μεταξύ τους. Μόλις ενεργοποιηθεί το workflow, το GitHub Actions ξεκινά την εκτέλεσή τους ταυτόχρονα.

Οι έλεγχοι αυτοί χωρίζονται σε δύο κατηγορίες, στατικούς ελέγχους, οι οποίοι αναλύουν τον κώδικα και τις εξαρτήσεις χωρίς να απαιτείται η εκτέλεση της εφαρμογής, και δυναμικούς ελέγχους, οι οποίοι πραγματοποιούνται αφού η δοκιμαστική εφαρμογή τεθεί σε λειτουργία. Μετά την ολοκλήρωση όλων αυτών των jobs, τα αποτελέσματα συγκεντρώνονται στο τελικό στάδιο παραγωγής της αναφοράς από τον AI Agent.

3.3 Στατικοί έλεγχοι

Οι στατικοί έλεγχοι δεν απαιτούν την εκτέλεση της εφαρμογής. Αντίθετα, τα εργαλεία εξετάζουν το repository για εκτεθειμένα μυστικά, προσωπικά δεδομένα, λογικές ευπάθειες στον κώδικα και γνωστά προβλήματα ασφαλείας σε βιβλιοθήκες τρίτων. Για τον σκοπό αυτό χρησιμοποιούνται τέσσερις έλεγχοι: Gitleaks, Custom PII Scanner, CodeQL και Trivy. Ακολουθεί η ανάλυση των βημάτων στατικών ελέγχων, χωρισμένη στα τέσσερα αυτά εργαλεία.

3.3.1 Σάρωση Μυστικών με το Gitleaks

Ο ρόλος ελέγχου σάρωσης μυστικών είναι ο εντοπισμός εκτεθειμένων μυστικών εντός του πηγαίου κώδικα, όπως API keys, κωδικοί πρόσβασης και authentication tokens. Η διαρροή τέτοιων δεδομένων είναι επικίνδυνη, γιατί μπορούν να χρησιμοποιηθούν για τη μη εξουσιοδοτημένη πρόσβαση σε βάσεις δεδομένων ή cloud υποδομές του οργανισμού. Η υλοποίηση γίνεται με το εργαλείο Gitleaks, το οποίο έχει σχεδιαστεί ειδικά για να αναγνωρίζει μοτίβα τέτοιων κωδικών.

```
secret_scan:
  name: Secret Scan (Gitleaks)
  runs-on: ubuntu-latest
  steps:
    - uses: actions/checkout@v4
      with:
        fetch-depth: 0
    - name: Run Gitleaks Scan
      uses: gitleaks/gitleaks-action@v2
      continue-on-error: true
      env:
        GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
      with:
        report-format: sarif
        report-path: gitleaks-report.sarif
    - uses: actions/upload-artifact@v4
      with:
        name: gitleaks-report
        path: gitleaks-report.sarif
```

Η συνολική ροή του job ακολουθεί τα εξής βήματα:

- **Checkout:** Λήψη του πηγαίου κώδικα του repository.
- **Run:** Εκτέλεση του έτοιμου GitHub Action gitleaks/gitleaks-action@v2, το οποίο σαρώνει τον κώδικα.
- **Artifact:** Τα ευρήματα αποθηκεύονται στο αρχείο gitleaks-report.sarif.

Επεξήγηση βασικών παραμέτρων του κώδικα:

- fetch-depth: 0: το Gitleaks αποκτά πρόσβαση σε ολόκληρο το ιστορικό του Git,
- continue-on-error: true: Επιτρέπει στο σύστημα ελέγχου να συνεχίσει την εκτέλεσή του ακόμα και αν το Gitleaks αποτύχει. Αυτό εξασφαλίζει ότι ο AI Agent λαμβάνει πάντα τα αποτελέσματα και τα εξηγεί στον developer, αντί να διακόπτεται απότομα η διαδικασία.

3.3.2 Σάρωση Ιδιωτικότητας με PII Scanner

Ο ρόλος του ελέγχου σάρωσης ιδιωτικότητας είναι ο εντοπισμός προσωπικά ταυτοποιήσιμων δεδομένων που μπορεί να βρίσκονται μέσα στον κώδικα. Η ύπαρξη τέτοιων δεδομένων είναι επικίνδυνη, επειδή εκθέτει τους χρήστες σε κινδύνους παραβίασης των ιδιωτικών τους στοιχείων, όπως είναι ο αριθμός τηλεφώνου, η ηλεκτρονική διεύθυνση ή ακόμα και ο τραπεζικός αριθμός IBAN. Για τον έλεγχο αυτό αναπτύχθηκε ένα εξειδικευμένο Python script (piiscanner.py)

```
pii_scan:
  name: Privacy Scan (PII)
  runs-on: ubuntu-latest
  steps:
    - uses: actions/checkout@v4
    - uses: actions/setup-python@v5
      with:
        python-version: "3.11"
    - name: Run PII Scanner
      continue-on-error: true
```

```
    run: python piiscanner/piiscanner.py

  - uses: actions/upload-artifact@v4

    with:

      name: pii-report

      path: pii-report.sarif
```

Η συνολική ροή του job ακολουθεί τα εξής βήματα:

- **Checkout:** Λήψη του πηγαίου κώδικα του repository.
- **Setup Python:** Εγκατάσταση περιβάλλοντος Python έκδοσης 3.11 στον runner.
- **Run:** Εκτέλεση της εντολής python piiscanner/piiscanner.py, η οποία σαρώνει τα αρχεία και τυπώνει τα ευρήματα.
- **Artifact:** Τα αποτελέσματα αποθηκεύονται στο αρχείο pii-report.sarif.

Επεξήγηση κώδικα piiscanner.py

- **Αναγνώριση Μοτίβων:** Χρησιμοποιεί Κανονικές Εκφράσεις για να εντοπίσει συγκεκριμένα μοτίβα κειμένου, όπως για την αναγνώριση ελληνικών αριθμών IBAN.
- **Κάλυψη Δεδομένων (Masking):** Εφαρμόζει ειδικές συναρτήσεις (π.χ. mask_email) για να αντικαταστήσει τα ευαίσθητα δεδομένα με αστερίσκους (π.χ. ph***@domain.com)..
- **Μορφοποίηση σε SARIF:** Μετατρέπει τα "καλυμμένα" ευρήματα σε ένα τυποποιημένο αρχείο SARIF JSON.

3.3.3 Στατική Ανάλυση Κώδικα με το CodeQL

Ο ρόλος της Στατικής Ανάλυσης είναι να εξετάζει τη δομή και τη λογική του πηγαίου κώδικα για ευπάθειες που θα μπορούσαν να οδηγήσουν σε επιθέσεις, επιτρέποντας σε κακόβουλους χρήστες να χειραγωγήσουν την εφαρμογή όταν αυτή εκτελείται. Για τον σκοπό αυτό αξιοποιείται το CodeQL, η επίσημη μηχανή ανάλυσης της πλατφόρμας του GitHub.

```
 sast_scan:

   name: Static Scan (CodeQL)
```

```
runs-on: ubuntu-latest
permissions:
  security-events: write
  packages: read
  actions: read
  contents: read
strategy:
  fail-fast: false
  matrix:
    include:
      - language: python
        build-mode: none
steps:
  - uses: actions/checkout@v4
  - uses: github/codeql-action/init@v4
    with:
      languages: ${{ matrix.language }}
      build-mode: ${{ matrix.build-mode }}
  - uses: github/codeql-action/analyze@v4
    with:
      category: "/language:${{matrix.language}}"
      output: codeql-results
  - uses: actions/upload-artifact@v4
    with:
      name: codeql-report
```

path: codeql-results/

Η συνολική ροή του job ακολουθεί τα εξής βήματα:

- **Checkout:** Λήψη του πηγαίου κώδικα του repository.
- **Initialize CodeQL:** Αρχικοποίηση της μηχανής του CodeQL βάσει της γλώσσας που έχει δηλωθεί.
- **Analyze:** Έναρξη της ανάλυσης
- **Artifact:** Τα ευρήματα αποθηκεύονται στο φάκελο codeql-results/ ως artifact.

Επεξήγηση βασικών παραμέτρων του κώδικα:

- permissions: security-events: write: Παρέχει στο Action το δικαίωμα εγγραφής, ώστε τα ευρήματα να μπορούν να γράφονται και να εμφανίζονται απευθείας στην καρτέλα του GitHub repository.
- strategy: matrix: Επιτρέπει τη δυναμική εκτέλεση του ελέγχου για πολλαπλές γλώσσες προγραμματισμού.
- fail-fast: false: Διασφαλίζει ότι αν αποτύχει η ανάλυση σε μία γλώσσα του matrix, δεν θα ακυρωθούν αυτόματα οι αναλύσεις των υπολοίπων γλωσσών.

3.3.4 Ανάλυση Εξαρτήσεων με το Trivy

Ο ρόλος της ανάλυσης εξαρτήσεων είναι να ελέγχει τα αρχεία εξαρτήσεων (όπως το requirements.txt ή το package.json) για βιβλιοθήκες που περιέχουν γνωστές και καταγεγραμμένες ευπάθειες. Η χρήση ευάλωτων εξαρτήσεων είναι επικίνδυνη, καθώς ακόμα και αν ο ίδιος ο κώδικας της εφαρμογής είναι απολύτως ασφαλής, το σύστημα μπορεί να κινδυνεύει από τις εξωτερικές βιβλιοθήκες που χρησιμοποιεί. Το έργο αυτό αναλαμβάνει το εργαλείο Trivy.

dependency_scan:

name: Dependency Scan (Trivy)

runs-on: ubuntu-latest

steps:

- uses: actions/checkout@v4

- name: Run Trivy FS Scan

uses: aquasecurity/trivy-action@0.35.0

```
    continue-on-error: true
    with:
      scan-type: fs
      scan-ref: .
      scanners: vuln
      severity: HIGH,CRITICAL
      format: sarif
      output: trivy-report.sarif
  - uses: actions/upload-artifact@v4
    with:
      name: trivy-report
      path: trivy-report.sarif
```

Η συνολική ροή του job ακολουθεί τα εξής βήματα:

- **Checkout:** Λήψη του πηγαίου κώδικα του repository.
- **Run Trivy:** Εκτέλεση του action aquasecurity/trivy-action@0.35.0 πάνω στα αρχεία του repository.
- **Artifact:** Τα ευρήματα αποθηκεύονται στο αρχείο trivy-report.sarif.

Επεξήγηση βασικών παραμέτρων του κώδικα:

- scan-type: fs: Δηλώνει στο Trivy να πραγματοποιήσει έλεγχο στο τοπικό σύστημα αρχείων του repository, αντί να ελέγξει κάποιο έτοιμο Docker image.
- scan-ref: Δηλώνει ότι η σάρωση θα ξεκινήσει από τον ριζικό φάκελο (.) του repository.
- scanners: vuln: Περιορίζει τη λειτουργία του Trivy αυστηρά στην αναζήτηση ευπαθειών
- severity: HIGH,CRITICAL: Φιλτράρει τα αποτελέσματα ώστε να καταγράφονται μόνο οι ευπάθειες υψηλού και κρίσιμου κινδύνου.

3.4 Δυναμικοί έλεγχοι

Σε αντίθεση με τους στατικούς ελέγχους, ο δυναμικός έλεγχος απαιτεί η εφαρμογή να εκτελείται ώστε να εντοπιστούν ευπάθειες που προκύπτουν αποκλειστικά κατά τη λειτουργία της. Ο δυναμικός έλεγχος της εργασίας, χωρίζεται σε δύο ανεξάρτητους ελέγχους: έναν γενικό έλεγχο ασφαλείας με το OWASP ZAP και έναν στοχευμένο έλεγχο για βάσεις δεδομένων με το SQLMap.

Πριν από την παρουσίαση των δύο εργαλείων, είναι απαραίτητο να εξηγηθεί το αρχικό στάδιο του dynamic_scan job, στο οποίο η demo εφαρμογή εγκαθίσταται, εκκινείται και ελέγχεται ότι ανταποκρίνεται. Συγκεκριμένα, η εφαρμογή λειτουργεί ως ο στόχος πάνω στον οποίο θα εκτελεστούν το OWASP ZAP και το SQLMap.

3.4.1 Προετοιμασία και Εκκίνηση της Demo Εφαρμογής

Στο πρώτο στάδιο του dynamic_scan job γίνεται η προετοιμασία του περιβάλλοντος εκτέλεσης. Επειδή οι δυναμικοί έλεγχοι χρειάζονται μια εφαρμογή που να τρέχει σε πραγματικό χρόνο, το workflow εγκαθιστά τις απαραίτητες εξαρτήσεις, εκκινεί τη demo εφαρμογή στο παρασκήνιο και στη συνέχεια περιμένει μέχρι να επιβεβαιωθεί ότι η εφαρμογή είναι διαθέσιμη στο τοπικό περιβάλλον του runner.

```
dynamic_scan:
  name: Dynamic Scan (ZAP + SQLMap)
  runs-on: ubuntu-latest
  steps:
    - uses: actions/checkout@v4
    - uses: actions/setup-python@v5
      with:
        python-version: "3.11"
    - name: Install & Start Demo App
      working-directory: demo-app
      run: |
        pip install -r requirements.txt
        nohup python app.py > app.log 2>&1 &
    - name: Wait for demo app
      run: |
        for i in {1..15}; do
          if curl -s "http://localhost:5000/product?id=1" > /dev/null; then exit 0; fi
          sleep 2
        done
        exit 1
```

Η συνολική ροή του job ακολουθεί τα εξής βήματα:

- **Start Demo App:** Εγκατάσταση των απαραίτητων εξαρτήσεων και εκκίνηση της δοκιμαστικής εφαρμογής (Demo App) στο παρασκήνιο.
- **Wait for demo app:** Εκτέλεση ενός loop αναμονής, ώστε να διασφαλιστεί ότι η εφαρμογή έχει φορτώσει και ανταποκρίνεται, πριν ξεκινήσουν οι επιθέσεις.

Επεξήγηση βασικών παραμέτρων του κώδικα:

- nohup ... &: Επιτρέπει στην Python εφαρμογή να τρέχει στο background, με το GitHub Actions να προχωράει στα επόμενα βήματα των scans.

Μετά την επιβεβαίωση ότι η εφαρμογή είναι διαθέσιμη, το ίδιο job συνεχίζει με την εκτέλεση των δύο εργαλείων δυναμικής ανάλυσης OWASP ZAP και SQLMap.

3.4.2 Γενική Σάρωση Web Εφαρμογής με OWASP ZAP

Ο ρόλος αυτού του ελέγχου είναι η συνολική αξιολόγηση της ασφάλειας μιας web εφαρμογής ενώ αυτή βρίσκεται σε λειτουργία. Αναζητά κενά ασφαλείας στο επίπεδο του δικτύου και του HTTP, όπως κακές ρυθμίσεις, απουσία headers ασφαλείας και ευπάθειες τύπου XSS. Η ύπαρξη τέτοιων προβλημάτων είναι επικίνδυνη, καθώς επιτρέπουν σε επιτιθέμενους να παρακάμψουν μηχανισμούς προστασίας του browser. Η υλοποίηση γίνεται με το OWASP ZAP, το οποίο αναλαμβάνει να στείλει αυτοματοποιημένα HTTP αιτήματα στην εφαρμογή, με σκοπό να εντοπίσει πιθανά προβλήματα ασφάλειας που εμφανίζονται μόνο κατά την εκτέλεσή της.

```
- name: Run ZAP Full Scan

  run: |
    mkdir -p ZAP-reports
    chmod 777 ZAP-reports
    docker run --rm --network host -v ${{ github.workspace }}/ZAP-reports:/ZAP/wrk/:rw -t
ghcr.io/ZAProxy/ZAProxy:stable ZAP-full-scan.py -t http://localhost:5000 -I -J ZAP-report.json || true
```

Στο συγκεκριμένο workflow, το OWASP ZAP δεν εγκαθίσταται απευθείας στον GitHub Actions runner. Αντίθετα, αξιοποιεί ένα έτοιμο Docker image, δημιουργώντας ένα προσωρινό container με σταθερό περιβάλλον σε κάθε εκτέλεση.

Η διαδικασία ελέγχου ακολουθεί τα εξής βήματα:

- **Εκκίνηση Εφαρμογής:** Το workflow εκκινεί τη demo εφαρμογή στο `http://localhost:5000`.
- **Ενεργοποίηση ZAP:** Το Docker container του ZAP ξεκινά και εκτελεί το script `ZAP-full-scan.py`, στοχεύοντας την τοπική εφαρμογή.
- **Ενεργός Έλεγχος:** Το ZAP εξερευνά τις διαθέσιμες σελίδες και τα endpoints για να δημιουργήσει την εικόνα της επιφάνειας επίθεσης και στέλνει ειδικά διαμορφωμένα HTTP αιτήματα για να ελέγξει ενεργά την εφαρμογή και να εντοπίσει κενά ασφαλείας.

Επεξήγηση βασικών παραμέτρων του κώδικα:

- --network host: Μια ρύθμιση του Docker, η οποία επιτρέπει στο container του ZAP να έχει άμεση πρόσβαση στο τοπικό δίκτυο του runner, προκειμένου να μπορεί να "δει" και να αλληλεπιδράσει με το http://localhost:5000.
- -J ZAP-report.json: Τα αποτελέσματα του ZAP αποθηκεύονται σε μορφή JSON.
- || true: Αν το ZAP βρει κρίσιμο πρόβλημα και επιστρέψει κωδικό σφάλματος, το step δεν θα σταματήσει.

3.4.3 Στοχευμένη Σάρωση SQL Injection με SQLMap

Ο ρόλος αυτού του ελέγχου είναι ο εξειδικευμένος εντοπισμός και η επιβεβαίωση ευπαθειών τύπου SQL Injection σε συγκεκριμένα σημεία της εφαρμογής, μέσω των οποίων ο χρήστης περνάει δεδομένα. Η ύπαρξη τέτοιων ευπαθειών είναι επικίνδυνη, καθώς επιτρέπει σε έναν επιτιθέμενο να παρακάμψει τη λογική της εφαρμογής και να διαβάσει, να αλλάξει ή και να διαγράψει απευθείας τα δεδομένα της βάσης δεδομένων. Η υλοποίηση γίνεται με το SQLMap.

```
- name: Run SQLMap Scan

  run: |

    mkdir -p sqlmap-reports

    docker run --rm --network host -v ${{ github.workspace
}}/sqlmap-reports:/root/.local/share/sqlmap/output parrotsec/sqlmap -u
"http://localhost:5000/product?id=1" --batch --answers="follow=Y,reduce=Y"
--output-dir=/root/.local/share/sqlmap/output | tee sqlmap-reports/sqlmap-output.txt || true

- uses: actions/upload-artifact@v4

  with:
```

name: dynamic-reports

path: |

ZAP-reports/

sqlmap-reports/

Η συνολική ροή του job ακολουθεί τα εξής βήματα:

- **Run SQLMap:** Εκτέλεση του εργαλείου SQLMap μέσω Docker container, το οποίο στοχεύει το συγκεκριμένο URL (/product?id=1) αναζητώντας ευπάθειες στη βάση δεδομένων.
- **Artifact:** Τα ευρήματα από το ZAP και το SQLMap αποθηκεύονται μαζί ως ένα κοινό artifact με το όνομα dynamic-reports.

Επεξήγηση βασικών παραμέτρων του κώδικα:

- --batch: Ενεργοποιεί τη μη διαδραστική λειτουργία του SQLMap.
- --answers="follow=Y,reduce=Y": Παρέχει προκαθορισμένες θετικές απαντήσεις στις τυπικές ερωτήσεις που κάνει το SQLMap κατά την ανάλυση.
- || true: Αν το SQLMap βρει κρίσιμο πρόβλημα και επιστρέψει κωδικό σφάλματος, το step δεν θα σταματήσει.

Η ολοκλήρωση του SQLMap αποτελεί και το τελευταίο στάδιο του dynamic_scan job. Στο τέλος, τα αποτελέσματα και των δύο εργαλείων αποθηκεύονται ως artifact, ώστε να είναι διαθέσιμα στο τελικό job του συστήματος, όπου θα συγκεντρωθούν μαζί με τα ευρήματα των στατικών ελέγχων και θα χρησιμοποιηθούν για τη δημιουργία της τελικής αναφοράς από τον AI Agent.

3.5 Agent Τεχνητής Νοημοσύνης

Μόλις ολοκληρωθούν και τα πέντε αρχικά ανεξάρτητα jobs, ενεργοποιείται το τελικό βήμα το οποίο ονομάζεται ai_report. Αυτό το job είναι υπεύθυνο για τη συγκέντρωση και την επεξεργασία των δεδομένων, γι' αυτό και πρέπει να περιμένει να ολοκληρωθούν όλα τα προηγούμενα βήματα.

Ο ρόλος του συγκεκριμένου βήματος είναι σημαντικός, καθώς επιλύει το πρόβλημα της δυσανάγνωστης πληροφορίας. Αντί ο χρήστης να πρέπει να κατεβάσει και να διαβάσει ξεχωριστά αρχεία SARIF, JSON και logs , αυτό το στάδιο ενοποιεί όλα τα ευρήματα και

χρησιμοποιεί ένα μοντέλο Τεχνητής Νοημοσύνης για να τα μετατρέψει σε μια κατανοητή αναφορά.

```
 ai_report:
    name: Aggregate & AI Report
    runs-on: ubuntu-latest
    needs: [secret_scan, pii_scan, sast_scan, dependency_scan, dynamic_scan]
    if: always()
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-python@v5
        with:
          python-version: "3.11"

      - name: Download all reports
        uses: actions/download-artifact@v4
        with:
          path: all-reports

      - name: Install Dependencies
        run: python -m pip install -U google-genai

      - name: Aggregate all scan findings
        run: python aggregator.py all-reports/

      - name: Generate AI report
        env:
          GEMINI_API_KEY: ${{ secrets.GEMINI_API_KEY }}
          GEMINI_MODEL: gemini-3.1-flash-lite
        run: python scripts/agent.py

      - name: Add report to job summary
        run: |
          echo "# Unified Security Report" >> $GITHUB_STEP_SUMMARY
          echo "" >> $GITHUB_STEP_SUMMARY
          cat agent-report.md >> $GITHUB_STEP_SUMMARY

      - name: Upload Final Report
        uses: actions/upload-artifact@v4
        with:
          name: final-ai-report
          path: agent-report.md
```

Η συνολική ροή του job ακολουθεί τα εξής βήματα:

- Λήψη όλων των artifacts που παρήγαγαν τα προηγούμενα βήματα και αποθήκευσή τους σε έναν κοινό φάκελο (all-reports).
- Προετοιμασία της Python 3.11 και εγκατάσταση της βιβλιοθήκης google-genai για την επικοινωνία με το μοντέλο.
- Εκτέλεση του script aggregator.py, το οποίο διαβάζει τα διαφορετικά reports και τα οργανώνει σε ένα ενιαίο αρχείο.
- Εκτέλεση του scripts/agent.py, το οποίο στέλνει τα ενοποιημένα δεδομένα στο LLM και λαμβάνει την τελική αναφορά σε μορφή Markdown.
- Ενσωμάτωση της τελικής αναφοράς απευθείας στο περιβάλλον του GitHub Actions και ταυτόχρονη αποθήκευση της ως τελικό artifact.

Επεξήγηση βασικών παραμέτρων του κώδικα:

- needs: [secret_scan, ...]: Το GitHub Actions δεν θα ξεκινήσει αυτό το βήμα αν δεν έχουν τελειώσει τη λειτουργία τους τα 5 αυτόνομα εργαλεία σάρωσης.
- if: always(): Η συνθήκη always() αναγκάζει το ai_report να εκτελεστεί ακόμα και αν κάποιο από τα προηγούμενα jobs αποτύχει.
- run: python aggregator.py all-reports/: Το script αυτό λειτουργεί ως μεταφραστής. Διαβάζει τα SARIF αρχεία , τα JSON του ZAP και τα text outputs του SQLMap και τα μετατρέπει σε μια κοινή και καθαρή μορφή JSON, ώστε να μπορούν να εισαχθούν με σωστή μορφή στον Agent.
- GEMINI_MODEL: gemini-3.1-flash-lite: Καθορίζει το μοντέλο της Google που θα χρησιμοποιηθεί.

3.5.1 Κανονικοποίηση Δεδομένων

Η εντολή run: python aggregator.py all-reports/ αποτελεί τον κρίσιμο μεταφραστή του συστήματος. Τα εργαλεία στατικής και δυναμικής ανάλυσης εξάγουν τεράστιο όγκο δεδομένων, σε διαφορετικές δομές (SARIF για τον κώδικα, JSON για το ZAP, απλό κείμενο για το SQLMap) και με διαφορετικές κλίμακες αξιολόγησης. Αν αυτά τα αρχεία δίνονταν απευθείας στον Agent, ο όγκος τους θα ξεπερνούσε τα όρια του μοντέλου (token limits) και θα προκαλούσε σύγχυση.

Το aggregator.py σαρώνει τον φάκελο όπου έχουν συγκεντρωθεί όλα τα αποτελέσματα all-reports/. Η βασική του δουλειά είναι να αντλήσει μόνο τα δεδομένα που χρειαζόμαστε. Συγκεκριμένα:

- Κρατάει μόνο την πηγή, το όνομα της ευπάθειας, τη σοβαρότητά της, την περιγραφή και την απόδειξη του προβλήματος.
- Αφαιρεί ετικέτες HTML που προσθέτει το ZAP στις περιγραφές του, κάνοντας το κείμενο καθαρό.
- Μεταφράζει τα διαφορετικά συστήματα αξιολόγησης σε μια κοινή κλίμακα (Critical, High, Medium, Low).

Στο παρακάτω απόσπασμα από τη συνάρτηση parse_sarif_report, φαίνεται πώς το script εξάγει τα πεδία από τον κώδικα ενός αρχείου SARIF και μετατρέπει την αξιολόγησή του σε κοινή μορφή:

```
def parse_sarif_report(file_path):
    findings = []
    try:
        with open(file_path, "r", encoding="utf-8") as f:
            data = json.load(f)
        for run in data.get("runs", []):
            for result in run.get("results", []):
                rule_id = result.get("ruleId", "Unknown Vulnerability")
                message = result.get("message", {}).get("text", "No description provided.")
                level = result.get("level", "warning").lower()

                severity_map = {
                    "error": "High",
                    "warning": "Medium",
                    "note": "Low",
                    "none": "Informational"
                }
                severity = severity_map.get(level, "Medium")

                # Δημιουργία του τελικού, καθαρού αντικειμένου
                findings.append({
                    "source": tool_name,
                    "name": rule_id,
                    "severity": severity,
                    "description": message,
                    "evidence": evidence
                })
```

Μετά την επεξεργασία όλων των αρχείων, το script εξάγει τα καθαρισμένα ευρήματα σε ένα ενιαίο αρχείο με το όνομα aggregated-findings.json, το οποίο είναι πλέον έτοιμο για να διαβαστεί από τον Agent Τεχνητής Νοημοσύνης.

3.5.2 Δημιουργία της Τελικής Αναφοράς από τον LLM

Έχοντας εξασφαλίσει μια καθαρή πηγή δεδομένων, ενεργοποιείται το script agent.py.

Κρίσιμα αποσπάσματα του κώδικα agent.py:

```
def build_prompt(findings):

    simplified_findings = [simplify_finding(f) for f in findings]
```

findings_json = json.dumps(simplified_findings, indent=2, ensure_ascii=False)

return f"""

Your role is a DevOps fix guidance assistant.

Your job is to create a clear Markdown fix report from security scanner findings.

Rules:

Use only the findings provided.

Do not invent vulnerabilities.

Do not add tools that are not mentioned in the project.

Focus on risk, evidence, and safe fix guidance.

Prioritize High findings first, then Medium findings.

Keep the explanation clear and simple.

Mention the scanner source.

Output only Markdown.

You must separate findings into two categories: "Dynamic Security Findings" and "Static Security Findings".

Dynamic sources include: "ZAP", "SQLMap".

Static sources include: "Trivy", "Gitleaks", "CodeQL", "Custom PII Scanner".

Create a fix report for these findings:

{findings_json}

Use this structure:

1. Fix Report

2. Executive Summary

3. Prioritized Dynamic Findings

4. Prioritized Static Findings

5. Suggested fix Order

```
"""

def call_gemini(prompt):

    api_key = os.getenv("GEMINI_API_KEY")

    client = genai.Client(api_key=api_key)


    response = client.models.generate_content(

        model=MODEL_NAME,

        contents=prompt

    )

    usage = response.usage_metadata

    token_footer = (

        f"\n\n---\n### 📊 AI Token Usage Metrics\n"

        f"- **Prompt Tokens:** `{usage.prompt_token_count}`\n"

        f"- **Output Tokens:** `{usage.candidates_token_count}`\n"

        f"- **Total Tokens:** `{usage.total_token_count}`\n")

    return response.text + token_footer
```

Σύντομη επεξήγηση του κώδικα agent.py:

- **Φιλτράρισμα και Απλοποίηση Δεδομένων:** Από το αρχείο με τα ευρήματα, διατηρεί μόνο τα Critical, High και Medium. Επίσης, αφαιρούνται τα περιττά πεδία από το JSON. Αυτό μειώνει τον όγκο των δεδομένων βοηθώντας το μοντέλο να επικεντρωθεί στους πραγματικούς κινδύνους.
- **Κατασκευή Prompt:** Η συνάρτηση `build_prompt` κατασκευάζει τις οδηγίες προς το μοντέλο, ακολουθόντας τις εξής τακτικές ώστε το αποτέλεσμα να είναι όσο πιο αξιόπιστο γίνεται:
    - **Ρόλος και Στόχος:** Ορίζει το επιθυμητό ύφος, λέγοντας στο μοντέλο ότι λειτουργεί ως "Βοηθός παροχής οδηγιών διόρθωσης DevOps ", και του αναθέτει τον ξεκάθαρο στόχο να δημιουργήσει μια αναφορά διόρθωσης.
    - Αυστηροί Κανόνες: Επιβάλλει ακριβείς κανόνες, όπως «Χρησιμοποίησε αποκλειστικά τα παρεχόμενα ευρήματα» και «Μην επινοείς ευπάθειες» , με

σκοπό την αποτροπή του φαινομένου των "παραισθήσεων". Διασφαλίζοντας ότι το μοντέλο τεχνητής νοημοσύνης δεν θα επινοήσει ανύπαρκτες ευπάθειες.
    - **Δεδομένα Εισόδου:** Παρέχει στο μοντέλο τα ευρήματα προς ανάλυση, ενσωματώνοντας απευθείας τη μεταβλητή `{findings_json}` μέσα στο κείμενο.
    - **Δομή Εξόδου:** Επιβάλλει μια προκαθορισμένη μορφή παρουσίασης σε Markdown. Με σκοπό η ανάλυση να είναι κατανοητή, να εξηγεί στον χρήστη τι ευρήματα κινδύνου βρέθηκαν, σε ποιο μέρος του κώδικα και πώς μπορούν να διορθωθούν.
- **Κλήση του Μοντέλου Gemini API:** Η συνάρτηση `call_gemini` αναλαμβάνει την επικοινωνία με το API της Google. Επίσης, γίνεται και η εξαγωγή των token που χρειάστηκαν , τα οποία προστίθενται στο τέλος της αναφοράς. Καταγράφοντας τα tokens, ο χρήστης γνωρίζει ανά πάσα στιγμή το λειτουργικό κόστος του συστήματος, καθώς και το αν τα δεδομένα που στέλνει πλησιάζουν στο όριο υπερφόρτωσης του μοντέλου.

Συνοπτικά, το συγκεκριμένο σύστημα ελέγχου αναλαμβάνει την οργάνωση των ελέγχων ασφαλείας, εκτελεί τα εργαλεία στατικής και δυναμικής ανάλυσης, συγκεντρώνει τα παραγόμενα reports και τα συνθέτει σε μία ενιαία αναφορά μέσω του AI Agent.

Κεφάλαιο 4: Δοκιμές, Αξιολόγηση και Αποτελέσματα

Έχοντας ολοκληρώσει την αρχιτεκτονική σχεδίαση και την υλοποίηση του συστήματος, το επόμενο βήμα είναι η πρακτική δοκιμή και αξιολόγησή της. Με κύριο στόχο τον έλεγχο αν το σύστημα λειτουργεί όπως έχει σχεδιαστεί, εντοπίζοντας επιτυχώς διαφορετικούς τύπους κινδύνων και παράγοντας μια κατανοητή τελική αναφορά μέσω του Agent τεχνητής νοημοσύνης.

Για να αξιολογηθεί ένα αυτοματοποιημένο σύστημα ασφαλείας, απαιτείται ένα συγκεκριμένο πλαίσιο δοκιμών, το οποίο επιβάλλει την τροφοδότηση του συστήματος με σκόπιμα ευάλωτο κώδικα, εικονικά μυστικά και γνωστές ευάλωτες εξαρτήσεις. Με αυτόν τον τρόπο, μπορούμε να επιβεβαιώσουμε με απόλυτη ακρίβεια ότι κάθε εργαλείο του συστήματος ανταποκρίνεται σωστά στα κενά ασφαλείας που έχει σχεδιαστεί να εντοπίζει.

4.1 Δεδομένα Δοκιμής

Για την εκτέλεση των στατικών ελέγχων, διαμορφώθηκε ο φάκελος tests/ στο repository, ο οποίος περιέχει επιλεγμένα αρχεία που λειτουργούν ως δοκιμή για τους ελέγχους. Τα αρχεία αυτά χωρίζονται σε υποφακέλους, με τον καθένα να στοχεύει σε μια συγκεκριμένη κατηγορία κινδύνου.

Δείγματα για Διαρροή Μυστικών (Gitleaks Testing)

Για την αξιολόγηση του Gitleaks, διαμορφώθηκε ο φάκελος tests/secrets/. Εκεί τοποθετήθηκε ένα αρχείο με όνομα test.txt, το οποίο περιέχει μια συμβολοσειρά που προσομοιώνει ένα AWS Access Key:

AWS_ACCESS_KEY_ID=AKIA1234567890ABCDEF

Με σκοπό να διαπιστωθεί αν το σύστημα θα αναγνωρίσει επιτυχώς το μοτίβο του κωδικού ως παραβίαση ασφαλείας και θα το αναφέρει

Δείγματα Προσωπικών Δεδομένων (Privacy/PII Testing)

Για την αξιολόγηση του Python PIIscanner που εντοπίζει προσωπικά δεδομένα, δημιουργήθηκε ο φάκελος tests/pii_tests/, ο οποίος περιέχει τρία διαφορετικά αρχεία κειμένου:

- email.txt: Περιέχει ψεύτικες διευθύνσεις email (test1@example.com).
- iban.txt: Περιέχει έναν ψεύτικο τραπεζικό αριθμό μορφής IBAN (GR1601101250000000012300695).
- phone.txt: Περιέχει έναν ψεύτικο αριθμό κινητού τηλεφώνου (6980604941).

Με αυτόν τον τρόπο ελέγχεται η απόδοση των Κανονικών Εκφράσεων (Regex) του εργαλείου πάνω σε διαφορετικούς τύπους ευαίσθητων δεδομένων.

Δείγματα Στατικού Κώδικα

Για την αξιολόγηση του CodeQL να εντοπίζει λογικές ευπάθειες, τοποθετήθηκαν στον φάκελο tests/code/ δείγματα πηγαίου κώδικα. Συγκεκριμένα δημιουργήθηκαν τα εξής αρχεία:

- **vulnerable.py**: Μια μικρή εφαρμογή Flask η οποία περιέχει μια κλασική ευπάθεια SQL Injection. Η μεταβλητή username ενσωματώνεται στο SQL ερώτημα χωρίς κανέναν έλεγχο, επιτρέποντας την εκτέλεση κακόβουλων εντολών στη βάση δεδομένων:

  ```
  from flask import Flask, request

  import sqlite3

  app = Flask(__name__)

  @app.route("/user")

  def user():

      username = request.args.get("name", "")

      conn = sqlite3.connect("test.db")
  ```

cursor = conn.cursor()

*query = "SELECT * FROM users WHERE name = '" + username + "'"*

cursor.execute(query)

return "done"

- Επιπλέον, στον ίδιο φάκελο υπάρχουν και τα αρχεία safe.py και safe.js, τα οποία περιέχουν ασφαλές κώδικα. Αυτά προστέθηκαν για να επιβεβαιωθεί ότι το σύστημα δεν παράγει ψευδώς θετικά αποτελέσματα σε καθαρό κώδικα.

Ευάλωτες Εξαρτήσεις (Trivy Testing)

Για την αξιολόγηση του Trivy, στον έλεγχο εξωτερικών βιβλιοθηκών, χρησιμοποιήθηκε ο φάκελος tests/deps/. Μέσα σε αυτόν υπάρχουν τα αρχεία διαχείρισης πακέτων package.json και package-lock.json. Στα αρχεία αυτά ορίστηκε η χρήση της βιβλιοθήκης lodash στην έκδοση 4.17.15:

"dependencies": {

"lodash": "^4.17.15"

}

Η συγκεκριμένη έκδοση είναι παλαιότερη και φέρει γνωστές καταγεγραμμένες ευπάθειες. Η προσθήκη της λειτουργεί ως την ιδανική δοκιμή για να επιβεβαιωθεί ότι το Trivy ελέγχει σωστά τις εξαρτήσεις του repository

Δεδομένα Δυναμικής Ανάλυσης: Η Δοκιμαστική Εφαρμογή (Demo App)

Για την αξιολόγηση των εργαλείων δυναμικής ανάλυσης, δηλαδή του OWASP ZAP και του SQLMap, αναπτύχθηκε μια μικρή διαδικτυακή εφαρμογή , η οποία βρίσκεται στον φάκελο demo-app/. Η εφαρμογή αυτή είναι γραμμένη σε Python χρησιμοποιώντας το framework Flask και βασίζεται σε μια τοπική βάση δεδομένων SQLite. Κατά τη διάρκεια του δυναμικού ελέγχου, η εφαρμογή εκκινείται στο παρασκήνιο και λειτουργεί ως ο στόχος πάνω στον οποίο εκτελούνται οι επιθέσεις.

Η εφαρμογή κατασκευάστηκε με συγκεκριμένες αδυναμίες για την αξιολόγηση των δυναμικών ελέγχων:

- **SQL Injection:** Για την αξιολόγηση του SQLMap, στο αρχείο app.py και συγκεκριμένα στο endpoint /product η εφαρμογή δέχεται παραμέτρους μέσω του URL, όπως για παράδειγμα το /product?id=1. Η τιμή της παραμέτρου id εισάγεται απευθείας στο ερώτημα SQL χωρίς καμία χρήση παραμετροποιημένων ερωτημάτων, δημιουργώντας μια εκμεταλλεύσιμη ευπάθεια.

  @app.route("/product")

  def product():

  product_id = request.args.get("id", "")

  # ...

  # Η παράμετρος product_id ενσωματώνεται κατευθείαν στο ερώτημα

  query = f"SELECT id, name FROM products WHERE id = {product_id}"

  # ...

  c.execute(query)

- **Επισφαλείς Ρυθμίσεις HTTP:** Για την αξιολόγηση της ικανότητας του OWASP ZAP να εντοπίζει προβλήματα στο επίπεδο των ρυθμίσεων δικτύου, προστέθηκε η συνάρτηση apply_insecure_defaults στο αρχείο app.py. Αυτή η συνάρτηση εκτελείται αυτόματα μετά από κάθε αίτημα και εισάγει σκόπιμα αδύναμες ρυθμίσεις στα HTTP responses:

  @app.after_request

  def apply_insecure_defaults(response):

  response.set_cookie("user_session", "demo_user_12345")

  response.headers["X-Powered-By"] = "Flask/3.0.3 Python/3.10"

  response.headers["Server"] = "Ubuntu/22.04 LTS"

  return response

- **Δομή Πλοήγησης:** Τέλος, στον υποφάκελο templates/ δημιουργήθηκαν βασικές HTML σελίδες index.html και products.html. Αυτές οι σελίδες περιέχουν εσωτερικούς HTML συνδέσμους, όπως για παράδειγμα Sample Product. Η ύπαρξη αυτών των συνδέσμων επιτρέπει στο OWASP ZAP να πλοηγηθεί αυτόματα στην

εφαρμογή, να ανακαλύψει το κρυφό, ευάλωτο endpoint /product και να ξεκινήσει τη σάρωσή του.

Μέσα από αυτόν τον συνδυασμό στατικών αρχείων δοκιμής και της σκόπιμα ευάλωτης demo εφαρμογής, διασφαλίζεται ότι ολόκληρο το σύστημα έχει το κατάλληλο υλικό για να αξιολογηθεί σε πραγματικές συνθήκες.

4.2 Αποτελέσματα Ελέγχων

Αφού τα δεδομένα δοκιμής τοποθετήθηκαν στο repository και η εφαρμογή εκκινήθηκε στο παρασκήνιο, το pipeline εκτελέστηκε αυτοματοποιημένα μέσω των GitHub Actions.

Τα αποτελέσματα επιβεβαιώνουν τη σωστή λειτουργία όλων των εργαλείων, παράγοντας τα αντίστοιχα reports σε μορφή SARIF, JSON και TXT. Ακολουθεί η ανάλυση των ευρημάτων ανά εργαλείο.

- **Σάρωση Μυστικών (Gitleaks):** Το Gitleaks σάρωσε επιτυχώς τα αρχεία του repository και εντόπισε το ψεύτικο κλειδί που είχε τοποθετηθεί στο tests/secrets/test.txt. Το κατέγραψε σωστά ως generic-api-key, αποδεικνύοντας ότι μπορεί να αναγνωρίσει μορφές κλειδιών πρόσβασης, όπως τα AWS keys και να αποτρέψει τη διαρροή τους. Όπως φαίνεται στο απόσπασμα από το παραγόμενο SARIF αρχείο, το Gitleaks καταγράφει τον κανόνα που παραβιάστηκε και το ακριβές αρχείο:

  ```
  {
    "message": {
      "text": "generic-api-key has detected secret for file .github/workflows/tests/secrets/test.txt..."
    },
    "ruleId": "generic-api-key",
    "locations": [
      {
        "physicalLocation": {
          "artifactLocation": {
            "uri": ".github/workflows/tests/secrets/test.txt"
  …
  }
  ```

- **Σάρωση Ιδιωτικότητας (Custom PII Scanner):** Το Python script εντόπισε επιτυχώς τις διευθύνσεις email, τον αριθμό τηλεφώνου και τον αριθμό IBAN, διαβάζοντας τα αρχεία του φακέλου tests/pii tests/. Όπως φαίνεται στο απόσπασμα από το παραγόμενο SARIF αρχείο, τα δεδομένα αποκρύφθηκαν σωστά, δημιουργώντας ασφαλή previews:

  ```
  {

    "ruleId": "email",
    "message": {
      "text": "match_preview: te***@example.com"
    },
  ```

```
  "locations": [
   {
     "physicalLocation": {
       "artifactLocation": {
         "uri": "tests/pii tests/email.txt"
…
}
```

- **Ανάλυση Εξαρτήσεων (Trivy):** Το Trivy, ελέγχοντας το αρχείο package-lock.json, εντόπισε την παλιά βιβλιοθήκη lodash και ανέφερε τις κρίσιμες ευπάθειες. Επίσης, ανέλυσε το requirements.txt της Python και εντόπισε γνωστές ευπάθειες στις εγκατεστημένες εκδόσεις των βιβλιοθηκών Flask και requests. Όπως φαίνεται στο απόσπασμα από το παραγόμενο SARIF αρχείο, το Trivy καταγράφει τον κανόνα που παραβιάστηκε και το ακριβές αρχείο:

```
  "ruleId": "CVE-2020-8203",
  "level": "error",
  "message": {
    "text": "Package: lodash\nInstalled Version: 4.17.15\nVulnerability CVE-2020-8203\nSeverity:
HIGH\nFixed Version: 4.17.19..."
  },
  "locations": [
    {
      "physicalLocation": {
        "artifactLocation": {
          "uri": "tests/deps/package-lock.json"
…
}
```

- **Στατική Ανάλυση Λογικής (CodeQL):** Η μηχανή του CodeQL εντόπισε την ευπάθεια py/sql-injection τόσο στο αρχείο vulnerable.py, όσο και στο app.py της demo εφαρμογής. Επίσης, αναγνώρισε κινδύνους όπως το Reflective XSS και την έκθεση πληροφοριών στο Stack Trace στην Flask εφαρμογή. Όπως φαίνεται στο απόσπασμα από το παραγόμενο SARIF αρχείο, το CodeQL αποθηκεύει με ακρίβεια τη γραμμή κώδικα όπου εντοπίστηκε το SQL Injection:

```
{
  "ruleId": "py/sql-injection",
  "message": {
    "text": "This SQL query depends on a [user-provided value](1)."
  },
  "locations": [
    {
      "physicalLocation": {
        "artifactLocation": {
          "uri": "demo-app/app.py"
```

```
      },
      "region": {
        "startLine": 59
      }
…
}
```

- **Γενική Σάρωση Web Εφαρμογής (OWASP ZAP):** Το ZAP κατέγραψε με επιτυχία όλες τις σκόπιμες ευπάθειες. Εντόπισε με υψηλό επίπεδο κινδύνου το SQL Injection στο endpoint /product?id=. Επίσης, εντόπισε ευπάθεια διαρροής πηγαίου κώδικα στο ίδιο endpoint. Αναγνώρισε και τις αδύναμες ρυθμίσεις των HTTP Headers που υπήρχαν. Τέλος, βρήκε το cookies που δημιουργήθηκε χωρίς τα απαραίτητα ασφαλή flags HttpOnly και SameSite. Όπως φαίνεται στο απόσπασμα από το παραγόμενο JSON αρχείο του ZAP, περιγράφει το SQL Injection και την ένδειξη που οδήγησε στον εντοπισμό του:

```
{

  "alert": "SQL Injection - SQLite",
  "riskcode": "3",
  "riskdesc": "High (Medium)",
  "instances": [
    {
      "uri": "http://localhost:5000/product?id=%3B",
      "method": "GET",
      "param": "id",
      "evidence": "near \";\": syntax error"
…
}
```

- **Εξειδικευμένη Σάρωση Βάσεων Δεδομένων (SQLMap):** Το SQLMap στοχεύοντας αποκλειστικά το /product?id=1, επιβεβαίωσε το πρόβλημα. Κατέγραψε ότι η παράμετρος id είναι δυναμική και μπορεί να παραβιαστεί. Επιπλέον, αναγνώρισε αυτόματα ότι το σύστημα διαχείρισης βάσης δεδομένων είναι η SQLite και παρήγαγε επιτυχημένα payloads, αποδεικνύοντας ότι μπορεί να γίνει πλήρης εκμετάλλευση της βάσης. Όπως φαίνεται στο απόσπασμα από το παραγόμενο αρχείο σε μορφή κειμένου, το SQLMap εξάγει τα ευρήματά του, επιβεβαιώνοντας τα επιτυχημένα payloads επίθεσης:

```
sqlmap identified the following injection point(s) with a total of 53 HTTP(s) requests:
---
Parameter: id (GET)
    Type: boolean-based blind
    Title: AND boolean-based blind - WHERE or HAVING clause
    Payload: id=1 AND 3379=3379
```

```
    Type: time-based blind
    Title: SQLite > 2.0 AND time-based blind (heavy query)
    Payload: id=1 AND 9152=LIKE(CHAR(65,66,67...
---
[14:51:21] [INFO] the back-end DBMS is SQLite
```

Συνολικά, η παρουσίαση αυτών των αποτελεσμάτων αποδεικνύει ότι τα αυτόνομα scanners του συστήματος εκτέλεσαν τον σκοπό τους. Ωστόσο, παρήγαγαν τεράστιο και δυσανάγνωστο όγκο δεδομένων, σε διαφορετικές δομές, τον οποίο καλείται να διαχειριστεί το επόμενο στάδιο του συστήματος.

4.3 Συγκέντρωση και Κανονικοποίηση Δεδομένων

Το προηγούμενο στάδιο της αξιολόγησης ανέδειξε ένα βασικό πρόβλημα. Τα εργαλεία ασφαλείας εντοπίζουν τους κινδύνους, αλλά παράγουν έναν τεράστιο και ανομοιογενή όγκο δεδομένων. Κάθε έλεγχος χρησιμοποιεί τη δική του δομή και περιλαμβάνει πολλές περιττές τεχνικές λεπτομέρειες.

Αυτή ακριβώς τη δυσκολία κλήθηκε να λύσει το aggregator.py, το οποίο σάρωσε τον φάκελο all-reports/ και εφάρμοσε τους κανόνες κανονικοποίησης στα δεδομένα, εξάγοντας το ενιαίο αρχείο aggregated-findings.json.

Για να γίνει κατανοητή η αξία αυτού του σταδίου, ας συγκρίνουμε την μορφή ενός ευρήματος πριν και μετά από την επίδραση του aggregator.

Πριν: Το JSON του OWASP ZAP

```
{
  "pluginid": "40018",
  "alert": "SQL Injection - SQLite",
  "riskcode": "3",
  "riskdesc": "High (Medium)",
  "desc": "<p>SQL injection may be possible.</p>",
  "instances": [
    {
      "uri": "http://localhost:5000/product?id=%3B",
      "method": "GET",
      "param": "id",
      "evidence": "near \";\": syntax error",
      "otherinfo": "RDBMS [SQLite] likely..."
    }
  ],
```

"solution": " Do not trust client side input... Escape all data... "
}

Μετά: Το `aggregated-findings.json`

[
{
"source": "ZAP",
"name": "SQL Injection - SQLite",
"severity": "High",
"description": "SQL injection may be possible.",
"evidence": "URI: http://localhost:5000/product?id=%3B | Parameter: id | Evidence: near \";\": syntax error"
}
]

Ο aggregator αφαιρεί την HTML, εξάγει τα απαραίτητα πεδία και παραδίδει ένα καθαρό JSON, ιδανικό για να διαβαστεί από τον LLM Agent στο επόμενο βήμα.

4.4 Αξιολόγηση της Αναφοράς του Agent

Στο τελικό στάδιο του workflow το μοντέλο Τεχνητής Νοημοσύνης Gemini αναλαμβάνει να αναλύσει τα προετοιμασμένα ευρήματα του αρχείου aggregated-findings.json και να παράγει την τελική αναφορά διόρθωσης. Σκοπός αυτού του βήματος δεν είναι η εύρεση νέων προβλημάτων, αλλά η επεξήγηση των υπαρχόντων με τρόπο που να είναι άμεσα αξιοποιήσιμος από τον developer.

Η αναφορά ξεκινά με μια σύντομη περίληψη, η οποία συνοψίζει άμεσα τους μεγαλύτερους κινδύνους της εφαρμογής, όπως το SQL Injection και τις απαρχαιωμένες εξαρτήσεις, δίνοντας στον αναγνώστη τη γενική εικόνα του προβλήματος.

Αναλύοντας το παραγόμενο αποτέλεσμα, επιβεβαιώνεται ότι το μοντέλο τήρησε αυστηρά τους κανόνες που του δόθηκαν μέσω του prompt στο agent.py:

- **Σωστός Διαχωρισμός:** Το μοντέλο κατηγοριοποίησε επιτυχώς τα ευρήματα σε "Prioritized Dynamic Findings" και "Prioritized Static Findings"
- **Προτεραιοποίηση Κινδύνου:** Τήρησε τον κανόνα της προτεραιότητας, τοποθετώντας τα ευρήματα υψηλού κινδύνου, όπως το SQL Injection και τις ευάλωτες εξαρτήσεις, πάνω από τα ευρήματα μεσαίου κινδύνου, όπως η έλλειψη Security Headers και τα Hardcoded Secrets.
- **Αποφυγή Παραισθήσεων:** Πολύ σημαντικό είναι ότι το μοντέλο δεν επινόησε καμία ανύπαρκτη ευπάθεια και δεν πρότεινε εργαλεία που δεν υπήρχαν στο project. Βασίστηκε

αποκλειστικά στα δεδομένα που του παρασχέθηκαν, αναφέροντας με ακρίβεια τις τοποθεσίες των σφαλμάτων.

Η πραγματική αξία του Agent αναδεικνύεται από τον τρόπο που παρουσιάζει την πληροφορία. Αντί ο developer να χρειάζεται να διαβάσει χαώδη logs για να κατανοήσει τι πήγε στραβά, λαμβάνει μια ξεκάθαρη, δομημένη λίστα. Για κάθε εύρημα, ο Agent:

- Αναφέρει το πρόβλημα.
- Αναλύει γιατί είναι σημαντικό, που εξηγεί τον πραγματικό κίνδυνο που διατρέχει το σύστημα.
- Προτείνει διορθώσεις, όπου προσφέρει συγκεκριμένες και άμεσα εφαρμόσιμες τεχνικές λύσεις.

Η αναφορά ολοκληρώνεται με την ενότητα **"**Προτεινόμενη Σειρά Διόρθωσης**"**. Πρόκειται για έναν πρακτικό οδηγό προτεραιοτήτων, ο οποίος ταξινομεί τα προβλήματα ξεκινώντας από αυτά Υψηλού κινδύνου και προχωρώντας στα ζητήματα Μεσαίου κινδύνου, οργανώνοντας έτσι τα επόμενα βήματα διόρθωσης με λογική σειρά.

Με την προσθήκη του Agent Τεχνητής Νοημοσύνης, η ροή ελέγχων ασφαλείας μετατρέπεται σε έναν ενεργό σύμβουλο ασφαλείας. Ο Agent αναλαμβάνει να συνδέσει την ανίχνευση της ευπάθειας με την τελική της επιδιόρθωση, εξηγώντας το πρόβλημα και δίνοντας στον χρήστη οδηγίες αποκατάστασης γραμμένες σε φυσική γλώσσα. Έτσι, η διόρθωση των ευπαθειών γίνεται πιο εύκολη και κατανοητή.

Κεφάλαιο 5: Σχετικές Εργασίες

Είναι σημαντικό να μελετήσουμε τι έχουν δείξει άλλες πρόσφατες έρευνες γύρω από τις δυνατότητες των εργαλείων στατικού ελέγχου, τους τρόπους βελτίωσής τους, αλλά και τη συμπεριφορά των ίδιων των προγραμματιστών όσον αφορά τις διαρροές μυστικών. Οι παρακάτω μελέτες μας βοηθούν να κατανοήσουμε καλύτερα τα προβλήματα που υπάρχουν σχετικά με τα παραπάνω ζητήματα.

5.1 Δυνατότητες των Εργαλείων SAST

Αυτή η κατηγορία εστιάζει στο πώς λειτουργούν και ποια είναι τα όρια των εργαλείων στατικού ελέγχου.

- **Vulnerabilities mapping based on OWASP-SANS: a survey for static application security testing (SAST) (Li, 2020)**: Η έρευνα αυτή εξέτασε διάφορα εργαλεία στατικού ελέγχου (SAST) και το πώς κατηγοριοποιούν τα προβλήματα. Έδειξε ότι υπάρχει μεγάλη

ανομοιογένεια, καθώς το κάθε εργαλείο «βλέπει» και αναφέρει τα κενά ασφαλείας με τον δικό του τρόπο, κάτι που συχνά μπερδεύει τις ομάδες ανάπτυξης.

- **Techniques of sast tools in the early stages of secure software development: A systematic literature review (Jerónimo et al., 2024)**: Η έρευνα αυτή εστίασε στα εργαλεία που τρέχουν αυτόματα κατά την ανάπτυξη λογισμικού. Η έρευνα απέδειξε ότι τα δύο μεγαλύτερα προβλήματά τους είναι τα πολλά ψευδή σφάλματα (false positives) και το ότι δεν καλύπτουν πάντα όλο τον κώδικα.
- **A Systematic Literature Review on Static Application Security Testing (SAST) Tools: Evaluation, Benchmarks, Challenges, and Future Directions (Dalaq et al., 2025)**: Η συγκεκριμένη έρευνα απέδειξε ότι τα εργαλεία SAST τα πηγαίνουν περίφημα στα δοκιμαστικά περιβάλλοντα, αλλά όταν μπαίνουν σε πραγματικό κώδικα, εντοπίζουν μόλις το 12.7% των πραγματικών ευπαθειών.

5.2 Εμπόδια Χρήσης και Βελτιώσεις των Εργαλείων

Αυτή η κατηγορία εξετάζει γιατί οι χρήστες δυσκολεύονται να χρησιμοποιήσουν τα εργαλεία και πώς μπορούν αυτά να γίνουν καλύτερα.

- **Barriers to using static application security testing (sast) tools: A literature review (Wadhams et al., 2024)**: Η συγκεκριμένη έρευνα έψαξε να βρει γιατί οι προγραμματιστές αποφεύγουν να χρησιμοποιούν εργαλεία ελέγχου κώδικα. Το βασικό συμπέρασμα ήταν ότι το μεγαλύτερο εμπόδιο δεν είναι το κόστος, αλλά ο υπερβολικός αριθμός άσχετων προειδοποιήσεων.
- **Semgrep*: Improving the limited performance of static application security testing (sast) tools (Bennett et al., 2024)**: Αυτή η έρευνα προσπάθησε να λύσει το πρόβλημα των περιορισμένων δυνατοτήτων που έχουν τα γρήγορα εργαλεία, εστιάζοντας στο Semgrep. Πρότειναν τρόπους για να γίνει το εργαλείο πιο ακριβές στο τι βρίσκει, χωρίς όμως να χάσει την ταχύτητά του.

5.3 Διαρροές Μυστικών και Συμπεριφορά Προγραμματιστών

Αυτή η κατηγορία αφορά το πρόβλημα της έκθεσης ευαίσθητων δεδομένων και των συνηθειών των προγραμματιστών γύρω από τους κωδικούς.

- **How bad can it git? characterizing secret leakage in public github repositories (Meli et al., 2019)**: Η συγκεκριμένη έρευνα έδειξε πόσο εύκολα και μαζικά διαρρέουν ευαίσθητα δεδομένα (όπως API keys) σε δημόσιες πλατφόρμες αποθήκευσης κώδικα (όπως το GitHub) . Η ομάδα έφτιαξε μηχανισμούς σάρωσης και απέδειξε ότι χιλιάδες μυστικά διαρρέουν καθημερινά, κάτι που δείχνει πόσο απαραίτητα είναι εργαλεία όπως το Gitleaks.
- **Tales from the Git: Automating the detection of secrets on code and assessing developers' passwords choices (Lykousas & Patsakis, 2023)**: Η έρευνα αυτή ανέλυσε κωδικούς που διέρρευσαν στο GitHub για να δει πώς διαχειρίζονται τους κωδικούς οι επαγγελματίες προγραμματιστές. Το συμπέρασμα ήταν ότι, παρόλο που οι προγραμματιστές κάνουν το λάθος να ανεβάζουν κωδικούς στον κώδικά τους,

τουλάχιστον επιλέγουν πιο ισχυρούς και πολύπλοκους κωδικούς από τους απλούς χρήστες.

- **Decoding developer password patterns: A comparative analysis of password extraction and selection practices (Lykousas & Patsakis, 2024)**: Ως συνέχεια της προηγούμενης έρευνας, οι ίδιοι ερευνητές χρησιμοποίησαν μηχανική μάθηση για να καταλάβουν τον τρόπο που οι προγραμματιστές φτιάχνουν τους κωδικούς τους. Αυτό βοηθάει στο να δημιουργηθούν καλύτερα συστήματα που θα εντοπίζουν τους αδύναμους κωδικούς μέσα στα αρχεία.

5.4 Σύνδεση με την Παρούσα Εργασία

Διαβάζοντας τις παραπάνω μελέτες, είναι εμφανές ότι ένα από τα μεγαλύτερα προβλήματα των εργαλείων ελέγχου είναι ο τεράστιος όγκος δυσνόητων προειδοποιήσεων καθώς και η μεγάλη ανομοιογένεια σχετικά με τον τρόπο που παρουσιάζονται τα ευρήματα. Η παρούσα εργασία αντιμετωπίζει αυτό το πρόβλημα, με έναν Agent Τεχνητής Νοημοσύνης στο τελικό στάδιο της ροής ελέγχων. Αντί ο χρήστης να έρχεται αντιμέτωπος με χαώδη τεχνικά αρχεία, ο Agent αναλαμβάνει να συγκεντρώσει, να φιλτράρει και να του εξηγήσει με απλά λόγια τα ευρήματα, προτείνοντας ξεκάθαρα βήματα διόρθωσης.

Επιπλέον, λαμβάνοντας υπόψη τον κίνδυνο της διαρροής κωδικών, η εργασία ενσωματώνει το εργαλείο Gitleaks, καθώς και έναν custom PII scanner, ώστε να αποτρέπει την έκθεση μυστικών και προσωπικών δεδομένων.

Τέλος, για να βελτιωθούν τα χαμηλά ποσοστά ανίχνευσης που έχουν από μόνα τους τα εργαλεία SAST σε πραγματικές συνθήκες, το σύστημά δεν περιορίζεται μόνο στη στατική ανάλυση, καθώς τη συνδυάζει με εργαλεία δυναμικής ανάλυσης τα οποία ελέγχουν την εφαρμογή την ώρα που τρέχει.

Κεφάλαιο 6: Σύνοψη

Ο βασικός στόχος της εργασίας ήταν ο σχεδιασμός και η υλοποίηση μιας αυτοματοποιημένης ροής ελέγχων ασφάλειας και ιδιωτικότητας. Το σύστημα συνδυάζει διαφορετικά εργαλεία ανάλυσης με σκοπό την έγκαιρη ανίχνευση, οργάνωση και αντιμετώπιση ευπαθειών πριν αυτές γίνουν επικίνδυνες για την ασφάλεια του κώδικα.

Το σύστημα ελέγχου κατάφερε να εκτελέσει επιτυχώς παράλληλους στατικούς ελέγχους στον πηγαίο κώδικα και δυναμικούς ελέγχους σε μια ενεργή εφαρμογή. Διαχειρίστηκε τον τεράστιο όγκο δεδομένων που παράγουν αυτά τα εργαλεία και αξιοποίησε τον Agent τεχνητής νοημοσύνης, με αποτέλεσμα τα πολύπλοκα τεχνικά αρχεία να μετατραπούν σε μια κατανοητή και πρακτική αναφορά διόρθωσης.

Παρά την επιτυχημένη λειτουργία της, η συγκεκριμένη εργασία παρουσιάζει περιορισμούς, οι οποίοι θα μπορούσαν να αποτελέσουν αντικείμενο περαιτέρω διερεύνησης και βελτίωσης σε ενδεχόμενη μελλοντική εξέλιξη.

Εύρος Εντοπισμού Ευπαθειών

- **Ο περιορισμός:** Στην τωρινή του μορφή, το σύστημα ελέγχει πολύ συγκεκριμένους κινδύνους: διαρροές μυστικών, PII δεδομένα, ευάλωτες εξαρτήσεις, γενικές ρυθμίσεις HTTP και SQL Injections. Όσον αφορά τη στατική ανάλυση του CodeQL, έχει ρυθμιστεί να ελέγχει αποκλειστικά κώδικα γραμμένο σε Python. Επομένως, δεν μπορεί να ελέγξει λογικές ευπάθειες σε άλλες γλώσσες, ούτε πολύπλοκα σφάλματα επιχειρηματικής λογικής. Αυτός ο περιορισμός επιλέχθηκε ώστε το project να διατηρήσει την απλότητά του και να επικεντρωθεί στην απόδειξη της αρχιτεκτονικής, χωρίς να υπερφορτωθεί με πολύπλοκες ρυθμίσεις ανά γλώσσα.
- **Η μελλοντική βελτίωση:** Η εργασία θα μπορούσε να εξελιχθεί με την προσθήκη δυναμικής αναγνώρισης γλωσσών προγραμματισμού (auto-language detection) στο CodeQL. Επιπλέον, θα μπορούσαν να προστεθούν εξειδικευμένα εργαλεία για ανάλυση υποδομής ως κώδικα και scanners που θα ελέγχουν πιο σύνθετα μοτίβα αδειοδότησης, μεγαλώνοντας το πεδίο προστασίας.

Δυναμική Ανάλυση σε Σύνθετα Εταιρικά Συστήματα

- **Ο περιορισμός:** Το τρέχον σύστημα δυναμικής ανάλυσης δεν θα μπορούσε να προστατεύσει αποτελεσματικά ένα πραγματικό και σύνθετο εταιρικό σύστημα. Ο λόγος είναι ότι το ZAP και το SQLMap εκτελούνται πάνω σε μια απλή, ελεγχόμενη εφαρμογή και στοχεύουν τυφλά ένα συγκεκριμένο endpoint. Σε ένα πραγματικό εταιρικό περιβάλλον, οι εφαρμογές απαιτούν σύνδεση, διαχείριση συνεδριών και περίπλοκη πλοήγηση. Το σύστημά μας δεν διαθέτει μηχανισμούς ταυτοποίησης για να προσπελάσει κλειδωμένες σελίδες.
- **Η μελλοντική βελτίωση:** Το σύστημα ελέγχου θα μπορούσε να αναβαθμιστεί με την ενσωμάτωση μηχανισμών Authenticated Scanning. Παρέχοντας ασφαλή δοκιμαστικά credentials στο ZAP μέσω GitHub Secrets, το εργαλείο θα μπορούσε να συνδέεται στην εφαρμογή και να σαρώνει το σύνολο των προστατευμένων λειτουργιών της. Παράλληλα, η προσθήκη σύγχρονων εργαλείων δυναμικού crawling, όπως το Selenium, θα βοηθούσε στην πλήρη ανάλυση πολύπλοκων Single Page Applications.

Εφαρμογή και Αξιοποίηση σε Αληθινά Repositories

- **Ο περιορισμός:** Για να μπορέσει το σύστημα να χρησιμοποιηθεί αποτελεσματικά σε ένα αληθινό repository από μια πραγματική ομάδα προγραμματιστών, λείπει η άμεση σύνδεσή της με τη διαδικασία ελέγχου κώδικα. Αυτή τη στιγμή, το workflow εκτελείται όταν γίνεται push στο main branch ή χειροκίνητα. Δεν ενεργοποιείται σε Pull Requests και δεν μπλοκάρει τη συγχώνευση του κώδικα αν βρεθούν κρίσιμα προβλήματα.

- **Η μελλοντική βελτίωση:** Το σύστημα μπορεί να εξελιχθεί αλλάζοντας τα triggers του GitHub Actions ώστε να τρέχει υποχρεωτικά σε κάθε Pull Request. Επιπλέον, ένα script θα μπορούσε να διαβάζει την αναφορά και αν εντοπίζει ευρήματα με σοβαρότητα "Critical" ή "High", να αποτυγχάνει σκόπιμα το pipeline.

Η εργασία αποδεικνύει ότι η ενσωμάτωση της τεχνητής νοημοσύνης σε μια αυτοματοποιημένη ροή ελέγχων ασφαλείας μπορεί να μεταμορφώσει τον τρόπο με τον οποίο προσεγγίζουμε τη διαχείριση ευπαθειών. Μετατρέποντας τα τεράστια τεχνικά δεδομένα σε πρακτικές και κατανοητές οδηγίες, το σύστημα ενισχύει την ασφάλεια της ανάπτυξης λογισμικού, κάνοντας την προστασία της ιδιωτικότητας και των δεδομένων μια πιο αποτελεσματική διαδικασία. Έτσι, το project πέτυχε τον σκοπό του, να βοηθά τον προγραμματιστή να εντοπίζει και να διορθώνει ευπάθειες γρήγορα και αποτελεσματικά, πριν ο κώδικας ενσωματωθεί στο κύριο branch.