การเพิ่มประสิทธิภาพ DevSecOps สำหรับความปลอดภัยแบบหลายชั้นใน CI/CD เวิร์คโฟลว์

นางสาววณิศรา คำราชา

วิทยานิพนธ์นี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตร
วิทยาศาสตรมหาบัณฑิต
สาขาวิชาการบริหารเครือข่ายและความมั่นคงปลอดภัยสารสนเทศ
บัณฑิตวิทยาลัย มหาวิทยาลัยเทคโนโลยีพระจอมเกล้าพระนครเหนือ
ปีการศึกษา 2567

การเพิ่มประสิทธิภาพ DevSecOps สำหรับความปลอดภัยแบบหลายชั้นใน CI/CD เวิร์คโฟลว์

นางสาววณิศรา คำราชา

วิทยานิพนธ์นี้เป็นส่วนหนึ่งของการศึกษาหลักสูตร
วิทยาศาสตรมหาบัณฑิต
สาขาวิชาการบริหารเครือข่ายและความมั่นคงปลอดภัยสารสนเทศ
บัณฑิตวิทยาลัย มหาวิทยาลัยเทคโนโลยีพระจอมเกล้าพระนครเหนือ
ปีการศึกษา 2567

ใบรับรองโครงร่างวิทยานิพนธ์

บัณฑิตวิทยาลัย มหาวิทยาลัยเทคโนโลยีพระจอมเกล้าพระนครเหนือ

เรื่อง การเพิ่มประสิทธิภาพ DevSecOps สำหรับความปลอดภัยแบบหลายชั้นใน CI/CD เวิร์คโฟลว์

โดย นางสาววณิศรา คำราชา

ได้รับอนุมัติให้นับเป็นส่วนหนึ่งของการศึกษาตามหลักสูตรวิทยาศาสตรมหาบัณฑิต สาขาวิชาการบริหารเครือข่ายและความมั่นคงปลอดภัยสารสนเทศ

คณบดีบัณฑิตวิทยาลัย / หัวหน้าภาควิชา

คณะกรรมการสอบวิทยานิพนธ์

..... ประธานกรรมการ

(สุมิตรา นวลมีศรี)

..... อาจารย์ที่ปรึกษา

(ผู้ช่วยศาสตราจารย์ ดร.สุนันฑา สดสี)

..... กรรมการ

(นวพร วิสิฐพงศ์พันธ์)

ชื่อ : นางสาววณิศรา คำราชา
ชื่อวิทยานิพนธ์ : การเพิ่มประสิทธิภาพ DevSecOps สำหรับความปลอดภัยแบบหลายชั้นใน CI/CD เวิร์คโฟลว์
สาขาวิชา : การบริหารเครือข่ายและความมั่นคงปลอดภัยสารสนเทศ
มหาวิทยาลัยเทคโนโลยีพระจอมเกล้าพระนครเหนือ
อาจารย์ที่ปรึกษาวิทยานิพนธ์หลัก : ผู้ช่วยศาสตราจารย์ ดร.สุนันฑา สดสี
ปีการศึกษา : 2567

บทคัดย่อ

งานวิจัยนี้นำเสนอแนวทางด้านความปลอดภัยแบบหลายชั้นสำหรับกระบวนการ Continuous Integration และ Continuous Delivery (CI/CD) เพื่อเพิ่มความปลอดภัย จุดประสงค์ของงานวิจัยนี้ คือเพื่อให้แนวทางในการลดช่องโหว่ในขั้นตอนการสร้าง (Build Stage) ซึ่งเป็นจุดสำคัญในการลดความเสี่ยงด้านความปลอดภัยตั้งแต่ระยะเริ่มต้นก่อนสภาพแวดล้อมการผลิตหรือก่อนที่จะใช้งานจริง อีกทั้งเพื่อวิเคราะห์ประสิทธิภาพของการใช้ทรัพยากรที่ Build Stage ใช้ รวมถึงทุกขั้นตอนของไปป์ไลน์ความปลอดภัยแบบหลายชั้นทั้งหมด โดยทรัพยากรที่วิเคราะห์ในงานวิจัยนี้ประกอบด้วย เวลาในการดำเนินการ (Execution Time) หน่วยความจำ (Memory) และ CPU การทดลองนี้ได้ใช้ Jenkins เพื่อทำการ Deploy แอปพลิเคชัน Node.js ที่มีขนาดและประเภทแตกต่างกัน ผลลัพธ์แสดงให้เห็นว่าหลังจากแก้ไขช่องโหว่ในขั้นตอนการสร้าง ช่องโหว่ลดลงถึงร้อยละ 16.5 โดยการใช้งาน CPU ถูกใช้อย่างเต็มประสิทธิภาพมากขึ้น ในขณะเดียวกัน การใช้หน่วยความจำของไปป์ไลน์ความปลอดภัยแบบหลายชั้นเพิ่มขึ้นในช่วงขั้นตอนสุดท้ายของกระบวนการ ซึ่งเพิ่มขึ้นเล็กน้อยเมื่อเทียบกับไปป์ไลน์ที่ไม่ปลอดภัย ใช้ Portainer เป็นเครื่องมือตรวจสอบทรัพยากร โดยสรุปแล้ว งานวิจัยแสดงให้เห็นว่าการบูรณาการความปลอดภัยเข้ากับ CI/CD ไปป์ไลน์ ไม่ส่งผลกระทบเชิงลบต่อทรัพยากร และสามารถให้ข้อมูลเชิงลึกเกี่ยวกับการใช้ทรัพยากรสำหรับวิศวกร DevOps เพื่อเป็นแนวทางปฏิบัติด้านความปลอดภัยไปใช้ใน CI/CD ไปป์ไลน์ได้อย่างมีประสิทธิภาพ

(มีจำนวนทั้งสิ้น 70 หน้า)

คำสำคัญ : เดฟเซคออปส์ การบูรณาการอย่างต่อเนื่องและการปรับใช้ ขั้นตอนการสร้างซอฟต์แวร์

_____ อาจารย์ที่ปรึกษาวิทยานิพนธ์หลัก

Name : Miss Wanisara Khamracha
Thesis Title : Enhancing Effectiveness of DevSecOps for Layered Security in CI/CD Workflows
Major Field : Digital Network and Information Security Management
King Mongkut's University of Technology North Bangkok
Thesis Advisor : Assistant Professor Dr.Sunantha Sodsee
Academic Year : 2024

ABSTRACT

This study presents a layered security approach for Continuous Integration and Continuous Delivery (CI/CD) pipelines to enhance security. The research aims to provide a guideline for reducing vulnerability in the build stage, which is the main point in reducing security risks from the beginning before the production environment, and to analyze the performance of resource usage that the build stage uses, as well as the whole stage of the layered secure pipeline. Herein, analyzed resources include Execution Time, Memory, and CPU. The experiment used Jenkins to deploy Node.js applications of different sizes and types. The results show that vulnerabilities in the build stage are reduced by up to 16.5% after remediation, while the CPU usage was more utilized efficiently. In parallel, the memory usage of the layered security pipeline was increased during the final stages of the pipeline, which is slightly higher than that of unsecured practices. Portainer, a lightweight monitoring tool, evaluates these resources for this work. Finally, this study shows that integrating security in the CI/CD pipeline does not negatively impact resource usage and provides insight into resource monitoring for DevOps engineers to effectively implement security practices within their CI/CD pipelines.

(Total 70 Pages)

Keywords: DevSecOps, CI/CD Pipeline, Snyk, Resource usage Analysis, Vulnerability Management, Build Stage Optimization

_____ Advisor

กิตติกรรมประกาศ

วิทยานิพนธ์ฉบับนี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญาวิทยาศาตรมหาบัณฑิต สำเร็จลุล่วงไปได้ด้วยความเมตตาและกรุณาอย่างยิ่งของ ผู้ช่วยศาสตราจารย์ ดร. สุนันฑา สดสี อาจารย์ที่ปรึกษาวิทยานิพนธ์ ที่ได้สละเวลาอันมีค่า ให้คำแนะนำที่มีประโยชน์ และให้กำลังใจตลอดระยะเวลาของการวิจัย ผู้วิจัยขอกราบขอบพระคุณเป็นอย่างสูง ไว้ ณ ที่นี้ด้วย

ขอขอบพระคุณ รองศาสตราจารย์ ดร. สุมิตรา นวลมีศรี และผู้ช่วยศาสตราจารย์ ดร. นวพร วิสิฐพงศ์พันธ์ กรรมการสอบวิทยานิพนธ์ ที่ได้สละเวลาอันมีค่า ในการตรวจสอบ ให้คำแนะนำข้อเสนอแนะ ในการปรับปรุงวิทยานิพนธ์ฉบับนี้ให้มีความสมบูรณ์มากยิ่งขึ้น

ขอขอบพระคุณ คุณพ่อธรรมศาสตร์ คำราชา คุณแม่นวรัตน์ คำราชา และญาติพี่น้องทุกคนที่ได้ให้การสนับสนุน ความรัก ความเป็นห่วง และเป็นกำลังใจที่ดีเสมอมา

สุดท้ายนี้ ผู้วิจัยขอขอบคุณ นายประกฤษฎ์ วงศ์นันทนานนท์ ที่คอยอยู่เคียงข้าง และให้คำปรึกษาในการทำวิจัยในครั้งนี้จนสำเร็จการศึกษา

วณิศรา คำราชา

สารบัญ

สารบัญ (ต่อ)

สารบัญตาราง

สารบัญภาพ

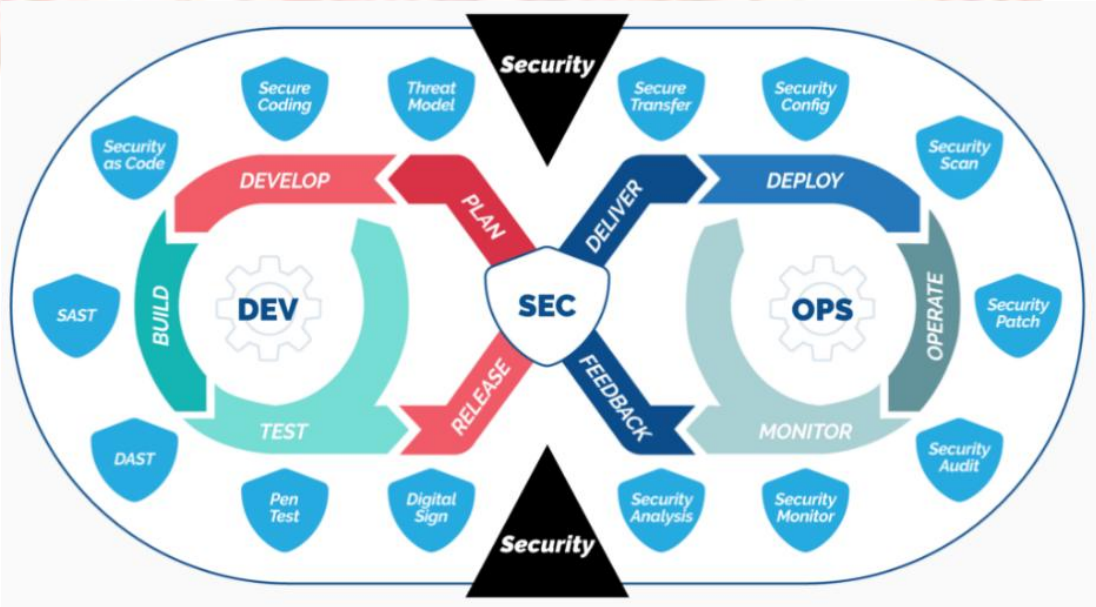
สารบัญภาพ (ต่อ)

สารบัญภาพ (ต่อ)

บทที่ 1
บทนำ

1.1 ความเป็นมาและความสำคัญของปัญหา

ด้วยการเติบโตอย่างรวดเร็วของเทคโนโลยี หลาย ๆ องค์กรจำเป็นต้องปรับกระบวนการทำงานและเร่งการส่งมอบซอฟต์แวร์ที่มีคุณภาพ เพื่อตอบสนองความต้องการของลูกค้า [1] ดังนั้นการนำแนวคิด DevOps มาปรับใช้เพื่อเพิ่มประสิทธิภาพในการพัฒนาซอฟต์แวร์ ที่เป็นการรวมกระบวนการทำงานระหว่างทีมพัฒนา (Development) กับทีมปฏิบัติการ (Operations) พร้อมทั้งนำระบบอัตโนมัติเข้ามาช่วยให้ทำงานได้อย่างต่อเนื่อง รวดเร็ว และมีประสิทธิภาพ ตามรายงานของ McKinsey รายงานว่าองค์กรที่ใช้ DevOps สามารถทำให้ผลิตภัณฑ์ซอฟต์แวร์เข้าสู่ตลาดได้เร็วขึ้นร้อยละ 30 ถึง 50 [2] ในขณะเดียวกัน จากการรายงานของ Cybersecurity Ventures สถานการณ์การโจมตีทางไซเบอร์คาดว่าต้นทุนความเสียหายจากอาชญากรรมทางไซเบอร์ทั่วโลกจะเพิ่มขึ้นร้อยละ 15 ต่อปีในช่วงสองปีข้างหน้าตั้งแต่ปี 2022 โดยค่าเสียหายอยู่ที่ 10.5 ล้านล้านเหรียญสหรัฐต่อปีภายในปี 2025 จากเดิมที่ 3 ล้านล้านเหรียญสหรัฐในปี 2015 [3] ทำให้การนำ DevSecOps มาใช้เป็นทางเลือกที่เหมาะสมที่สุดสำหรับสถานการณ์นี้



ภาพที่ 1-1 ขั้นตอนของวงจร DevSecOps [4]

เนื่องจากมาตรการความปลอดภัยจะถูกผสานอยู่ในทุกขั้นตอนของกระบวนการพัฒนาซอฟต์แวร์ ดังภาพที่ 1-1 ที่แสดงถึงการผสานมาตรการรักษาความปลอดภัยเข้าไปใน DevOps ทำให้กลายเป็น DevSecOps นอกจากนี้ วิธีหลักที่ใช้กันอย่างแพร่หลายของ DevOps ในการส่งมอบซอฟต์แวร์ให้กับลูกค้าอย่างรวดเร็ว คือ CI/CD (Continuous Integration/Continuous Delivery) pipeline เป็นกระบวนการที่ทำให้การพัฒนาซอฟต์แวร์เป็นไปโดยอัตโนมัติและมีประสิทธิภาพมากขึ้น รวมถึงการรวมและการเผยแพร่อย่างต่อเนื่องผ่านขั้นตอนต่าง ๆ มากมาย ประกอบด้วย ขั้นตอนการสร้างหรือขั้นตอนการสร้างซอฟต์แวร์ ขั้นตอนการทดสอบ และขั้นตอนการปรับใช้โค้ด [5] อีกทั้งยังสามารถรวมมาตรการรักษาความปลอดภัยหรือเครื่องมือรักษาความปลอดภัยได้ เช่น เครื่องมือตรวจจับและแก้ไขช่องโหว่เข้ากับไปป์ไลน์ CI/CD เพื่อลดความเสี่ยงด้านความปลอดภัยของการพัฒนาซอฟต์แวร์ได้ตั้งแต่เริ่มต้นก่อนที่จะเผยแพร่ [6] เนื่องจากภัยคุกคามของไปป์ไลน์ CI/CD ที่เกิดจากช่องโหว่มีอยู่มากมายและอาจเกิดขึ้นได้เสมอ เช่น การแทรกโค้ดจากการเขียนโปรแกรมที่ไม่ปลอดภัยและการอ้างอิงจากบุคคลที่สาม (Third-party Dependency) ที่เป็นอันตราย เพื่อจัดการกับปัญหานี้ ปัจจุบันจึงมีมาตรการและเครื่องมือรักษาความปลอดภัยรวมเข้าในแต่ละขั้นตอนของไปป์ไลน์ CI/CD [7] ยกตัวอย่างเช่น SonarQube เป็นเครื่องมือตรวจสอบคุณภาพของโค้ดในขั้นตอนการตรวจสอบโค้ด (Code Stage) Snyk เป็นเครื่องมือตรวจจับและแก้ไขช่องโหว่ของโค้ด Dependency และคอนเทนเนอร์อิมเมจ (Container Image) ในขั้นตอนการสร้าง (Build Stage) และ OWASP ZAP เป็นเครื่องมือเจาะระบบเพื่อทดสอบความปลอดภัยของเว็บแอปพลิเคชัน (Test Stage) จากขั้นตอนที่กล่าวมาทั้งหมด ขั้นตอนการสร้างมีความสำคัญในกระบวนการบูรณาการอย่างต่อเนื่อง (CI) [8] หากเพิ่มเครื่องมือความปลอดภัยในขั้นตอนนี้ในการตรวจจับหรือแก้ไขช่องโหว่ จะสามารถลดความเสี่ยงได้จริงก่อนที่จะถึงสภาพแวดล้อมการผลิต (Production Environment) [9] แต่งานศึกษาที่มีอยู่ยังไม่ได้ระบุว่าการผสานเครื่องมือความปลอดภัยของ CI/CD ไปป์ไลน์ ส่งผลกระทบต่อการใช้ทรัพยากรในไปป์ไลน์ CI/CD อย่างไร ดำเนินการโดยการตรวจสอบอย่างต่อเนื่อง (Monitoring) อีกทั้งยังถือว่าเป็นสิ่งสำคัญของ DevOps เพื่อให้ข้อมูลเชิงลึกเกี่ยวกับประสิทธิภาพทรัพยากรของเซิร์ฟเวอร์ที่ CI/CD ไปป์ไลน์ได้ใช้ โดยจะแสดงผลออกมาเป็นภาพเพื่อให้วิเคราะห์ได้อย่างง่ายดาย อย่างไรก็ตาม ทุก ๆ องค์กรควรใช้มาตรการรักษาความปลอดภัยในกระบวนการ CI/CD เพื่อรับประกันการส่งมอบและความปลอดภัยของซอฟต์แวร์ ซึ่งถือเป็นข้อได้เปรียบในโลกดิจิทัลที่เปลี่ยนแปลงอย่างรวดเร็ว ดังที่ได้กล่าวไปในตอนแรก

งานวิจัยนี้มีจุดมุ่งหมายเพื่อแก้ไขช่องโหว่ที่สำคัญในขั้นตอนการสร้าง (Build Stage) พร้อมตรวจสอบและวิเคราะห์การใช้ทรัพยากรก่อนและหลังการแก้ไขช่องโหว่ในขั้นตอนนี้ เพื่อประเมินผลกระทบของมาตรการด้านความปลอดภัย นอกจากนี้ งานวิจัยยังเปรียบเทียบการใช้ทรัพยากรของ CI/CD ไปป์ไลน์ที่ปลอดภัยแบบหลายชั้นทุกขั้นตอน (The Layered secure CI/CD Pipeline) และ CI/CD ไปป์ไลน์ที่ไม่มีความปลอดภัยใด ๆ (The Unsecured CI/CD Pipeline) เพื่อให้แนวทางปฏิบัติในการนำมาตรการความปลอดภัยมาใช้ใน CI/CD ไปป์ไลน์อย่างมีประสิทธิภาพ

1.2 วัตถุประสงค์ของการวิจัย

1.2.1 เพื่อนำเสนอแนวทางปฏิบัติด้านความปลอดภัยแบบหลายชั้นสำหรับ CI/CD ไปป์ไลน์

1.2.2 เพื่อนำเสนอข้อมูลเชิงลึกเกี่ยวกับการใช้ทรัพยากรของมาตรการความปลอดภัยที่นำมาปรับใช้ใน CI/CD ไปป์ไลน์

1.3 ขอบเขตงานวิจัย

1.3.1 ขั้นตอนของ CI/CD ไปป์ไลน์ที่ไม่มีความปลอดภัย (The Unsecured CI/CD Pipeline) จะประกอบด้วย ขั้นตอนการตรวจสอบโค้ด (Code Stage) ขั้นตอนการสร้าง (Build Stage) และขั้นตอนการปรับใช้ (Deployment Stage)

1.3.2 ขั้นตอนของ CI/CD ไปป์ไลน์ที่มีความปลอดภัยแบบหลายชั้น (The Layered secure CI/CD Pipeline) จะประกอบด้วย ขั้นตอนการตรวจสอบโค้ด (Code Stage) ขั้นตอนการสร้าง (Build Stage) ขั้นตอนการปรับใช้ (Deploy Stage) และขั้นตอนการทดสอบ (Test Stage)

1.3.3 เครื่องมือรักษาความปลอดภัยที่บูรณาการเข้ากับไปป์ไลน์ที่มีความปลอดภัยแบบหลายชั้นประกอบไปด้วย SonarQube ในขั้นตอนการตรวจสอบโค้ด Snyk และ Trivy ในขั้นตอนการสร้าง และ OWASP ZAP ในขั้นตอนการทดสอบ

1.3.4 ทำการแก้ไขช่องโหว่ที่ขั้นตอนการสร้าง (Build Stage) เท่านั้น

1.3.5 การปรับใช้ (Deployment) Docker Container จะรันอยู่บนเครื่องเซิร์ฟเวอร์เสมือนหรือคอมพิวเตอร์เสมือน (Virtual Machine) ที่เป็นระบบปฏิบัติการลินุกซ์ (Linux) สร้างโดยไมโครซอฟต์ อาซัวร์ (Microsoft Azure)

1.3.6 องค์ประกอบทางเทคนิคและเครื่องมือที่ใช้ในการทดลอง ใช้ Visual Studio Code เป็นสภาพแวดล้อมการพัฒนาหลักสำหรับการพัฒนา CI/CD Workflows และใช้ Jenkins โดยเป็นเครื่องมือที่ใช้รัน CI/CD ไปป์ไลน์ และใช้ GitHub เป็นเครื่องมือ Version Control และเก็บโค้ด

1.3.7 โค้ด (Source Code) ที่นำมาปรับใช้ (Deploy) พัฒนาด้วย Node.js โดยเป็น Front-end, Back-end และเว็บแอปพลิเคชัน (Web Application) จำนวน 9 ไฟล์โค้ด โดยมี 3 ขนาด ได้แก่ ขนาดเล็ก กลาง และใหญ่

1.3.8 ใช้ Docker ในการจัดการคอนเทนเนอร์ (Container)

1.3.9 การตรวจสอบทรัพยากรอย่างต่อเนื่อง (Resource Monitoring) ได้แก่ ทรัพยากรเวลาดำเนินการ (Execution Time) หน่วยความจำ (Memory) และการประมวลผล (CPU)

1.3.10 วิเคราะห์และเปรียบเทียบประสิทธิภาพการบูรณาการมาตรการความปลอดภัยของการทดลองนี้ใน 3 หัวข้อ ได้แก่

1.3.10.1 จำนวนช่องโหว่ก่อนและหลังแก้ไขใน Build Stage โดยจำแนกตามขนาดโค้ด

1.3.10.2 การใช้ทรัพยากรใน Build Stage ก่อนและหลังแก้ไขช่องโหว่ ได้แก่ เวลาดำเนินการ หน่วยความจำ และ CPU

1.3.10.3 เปรียบเทียบการใช้ทรัพยากรระหว่าง CI/CD ไปป์ไลน์ที่ไม่มีความปลอดภัย และ CI/CD ไปป์ไลน์ที่มีความปลอดภัยแบบหลายชั้น ได้แก่ เวลาดำเนินการ หน่วยความจำ และ CPU

1.4 วิธีการวิจัย

1.4.1 การจัดเตรียมสภาพแวดล้อมสำหรับการพัฒนาที่จะต้องใช้ในการทดลอง

1.4.1.1 สร้างเครื่องคอมพิวเตอร์เสมือนของไมโครซอฟต์ อาซัวร์ เวอร์ชัน Ubuntu 20.04 สเปก 2 vCPUs และ RAM 8 GB

1.4.1.2 ติดตั้ง Docker ลงในเครื่องคอมพิวเตอร์เสมือน สำหรับการจัดการคอนเทนเนอร์ (Container)

1.4.1.3 ติดตั้ง Jenkins ลงในคอมพิวเตอร์เสมือนสำหรับรัน CI/CD ไปป์ไลน์ทั้งสองแบบ และตรวจสอบเวลาดำเนินการของ CI/CD ไปป์ไลน์

1.4.1.4 ติดตั้ง Portainer ที่เป็นเครื่องมือจัดการ Container ลงในคอมพิวเตอร์เสมือน เพื่อเป็นเครื่องมือสำหรับการตรวจสอบทรัพยากรอย่างต่อเนื่อง (Resource Monitoring) ได้แก่ หน่วยความจำ (Memory) และการประมวลผล (CPU)

1.4.1.5 โค้ด (Source code) ที่จะนำมาปรับใช้ (Deploy) จำนวน 9 ไฟล์ ถูกเก็บไว้ใน GitHub ซึ่งเป็นเครื่องมือ Version Control

1.4.1.6 Visual Studio Code เป็นสภาพแวดล้อมในการพัฒนา CI/CD Workflows

1.4.2 การวางแผนพัฒนาทั้งสอง CI/CD เวิร์คโฟลว์ ใน Jenkins

1.4.2.1 การพัฒนา CI/CD ไปป์ไลน์ที่ไม่มีความปลอดภัย ประกอบด้วยขั้นตอน Code > Build > Deploy

1.4.2.2 การพัฒนา CI/CD ไปป์ไลน์ที่มีความปลอดภัยแบบหลายชั้น ประกอบด้วย Code > Build > Deploy > Test

1.4.2.2.1 บูรณาการเครื่องมือรักษาความปลอดภัยที่เหมาะสมกับแต่ละขั้นตอนลงใน CI/CD ไปป์ไลน์ ได้แก่ SonarQube สำหรับวิเคราะห์การพัฒนาโค้ดใน Code Stage, Snyk สำหรับตรวจจับและแก้ไขช่องโหว่ของ Dependency, Trivy สำหรับตรวจจับช่องโหว่ใน Build Stage และ OWASP ZAP สำหรับเจาะระบบเพื่อทดสอบความปลอดภัยของเว็บแอปพลิเคชันใน Test Stage

1.4.2.2.2 แก้ไขช่องโหว่ที่ Build Stage ด้วย Snyk สำหรับการรันไปป์ไลน์ครั้งที่หนึ่ง ตามด้วยตรวจจับช่องโหว่อีกครั้งหลังจากแก้ไขของ Snyk ด้วย Trivy สำหรับการรันไปป์ไลน์ครั้งที่สอง

1.4.3 การตรวจสอบประสิทธิภาพของทรัพยากร โดยใช้ Portainer ในการสังเกตแนวโน้มการใช้ทรัพยากรหน่วยความจำและ CPU ของไปป์ไลน์ทั้งสองรูปแบบ และ Build Stage ที่เป็นหนึ่งในขั้นตอนที่สำคัญของไปป์ไลน์

1.4.4 การวิเคราะห์และเปรียบเทียบประสิทธิภาพการบูรณาการมาตรการความปลอดภัย ดังนี้

1.4.4.1 จำนวนช่องโหว่ก่อนและหลังแก้ไขใน Build Stage

1.4.4.2 การใช้ทรัพยากรของ Build Stage ทั้งก่อนและหลังแก้ไขช่องโหว่

1.4.4.3 การใช้ทรัพยากรของ CI/CD ไปป์ไลน์ที่ไม่มีความปลอดภัยและ CI/CD ไปป์ไลน์ที่มีความปลอดภัยแบบหลายชั้น

1.5 นิยามศัพท์เฉพาะ

1.5.1 DevSecOps ย่อมาจาก การพัฒนา (Development) การรักษาความปลอดภัย (Security) และ การปฏิบัติการ (Operations) หมายถึง แนวทางปฏิบัติการบูรณาการการทดสอบความปลอดภัยในทุกขั้นตอนวงจรการพัฒนาซอฟต์แวร์

1.5.2 CI/CD ไปป์ไลน์ ย่อมาจาก Continuous Integration/Continuous Delivery หรือ Continuous Deployment หมายถึง วิธีการหนึ่งของ DevOps ที่เป็นชุดขั้นตอนอัตโนมัติในการพัฒนาซอฟต์แวร์ เพื่อให้ส่งมอบซอฟต์แวร์ได้อย่างรวดเร็วและมีคุณภาพ อีกทั้งเป็นแนวทางปฏิบัติในการพัฒนาซอฟต์แวร์ที่เน้นการทำงานร่วมกันและส่งมอบซอฟต์แวร์อย่างต่อเนื่อง

1.5.3 CI/CD ไปป์ไลน์ที่มีความปลอดภัยแบบหลายชั้น (The Layered secure CI/CD Pipeline) หมายถึง CI/CD ไปป์ไลน์ที่มีมาตรการรักษาความปลอดภัยหรือเครื่องมือความปลอดภัยรวมอยู่ในหลาย ๆ ขั้นตอน

1.5.4 CI/CD เวิร์คโฟลว์ (CI/CD Workflows) หมายถึงไฟล์ที่กำหนดขั้นตอนการทำงานอัตโนมัติ (Pipeline) ในระบบ CI/CD โดยทั่วไปไฟล์เหล่านี้จะอยู่ในรูปแบบ YAML (.yaml หรือ .yml) อีกทั้งจะ

แตกต่างกันไปตามแพลตฟอร์มที่รัน CI/CD ต่าง ๆ ยกตัวอย่างเช่น แพลตฟอร์ม GitHub Actions จะใช้ไฟล์ .github/workflows/*.yml และแพลตฟอร์ม Jenkins จะใช้ Jenkinsfile

1.5.5 ขั้นตอนการสร้าง (Build Stage) หรือขั้นตอนการสร้างซอฟต์แวร์ (Software) หมายถึง ขั้นตอนหนึ่งใน CI/CD ไปป์ไลน์ ที่รวบรวมโค้ด (Source code), การอ้างอิง (Dependency) และคอนเทนเนอร์อิมเมจ (Container Image) มาคอมไพล์ (Compile) เพื่อเตรียมสำหรับการปรับใช้ (Deployment)

1.5.6 การอ้างอิง (Dependency) หมายถึง ส่วนประกอบภายนอกที่จำเป็นสำหรับโปรเจกต์พัฒนาโค้ดให้สำเร็จ

1.5.7 Docker หมายถึง แพลตฟอร์มโอเพนซอร์ส (Open source) ที่ใช้ในการสร้าง ทดสอบ และติดตั้งแอปพลิเคชันอย่างง่ายและรวดเร็ว โดยใช้คอนเทนเนอร์ในการจำลองทรัพยากร ซึ่งแตกต่างจากคอมพิวเตอร์เสมือน โดยที่ Docker จะใช้เคอร์เนล (Kernel) ของระบบปฏิบัติการโฮสต์ร่วมกัน ทำให้ใช้ทรัพยากรน้อยกว่าและทำงานได้รวดเร็วกว่า

1.6 ประโยชน์ที่คาดว่าจะได้รับ

1.6.1. ได้แนวทางปฏิบัติของ CI/CD ไปป์ไลน์ที่ปลอดภัย ซึ่งคือ Layered secure Pipeline เพื่อเพิ่มความปลอดภัยให้กับทุกขั้นตอนและครอบคลุมในระยะยาว

1.6.2 ได้ข้อมูลเชิงลึกทางด้านการใช้ทรัพยากรให้กับวิศวกร DevOps (DevOps Engineer) หรือวิศวกรโครงสร้างพื้นฐาน (Infrastructure Engineer) เพื่อนำแนวทางปฏิบัติด้านความปลอดของ CI/CD ไปป์ไลน์ไปปรับใช้กับองค์กรของตน

บทที่ 2
เครื่องมือที่ใช้และงานวิจัยที่เกี่ยวข้อง

วิทยานิพนธ์นี้มีวัตถุประสงค์เพื่อแก้ไขช่องโหว่ที่สำคัญใน Build Stage ซึ่งเป็นหนึ่งในขั้นตอนของ CI/CD ไปป์ไลน์ พร้อมวิเคราะห์การใช้ทรัพยากร โดยผู้วิจัยได้ทำการค้นหาเครื่องมือต่าง ๆ และศึกษางานวิจัยที่เกี่ยวข้อง โดยได้แบ่งออกเป็นสองหัวข้อ ได้แก่ เครื่องมือที่ใช้ และงานวิจัยที่เกี่ยวข้อง มีรายละเอียดดังนี้

2.1 เครื่องมือที่ใช้

2.1.1 เครื่องมือสร้างและจัดการ CI/CD ไปป์ไลน์

2.1.1.1 Jenkins [10] คือ เซิร์ฟเวอร์โอเพนซอร์ส (Open source) สำหรับทีม DevOps ที่ช่วยให้การสร้าง การทดสอบ และการส่งมอบหรือปรับใช้ซอฟต์แวร์เป็นไปโดยอัตโนมัติ หรือกล่าวได้ว่า Jenkins เป็นเครื่องมือจัดการกระบวนการ CI/CD ที่พัฒนาด้วยภาษาจาวา (Java) และมีปลั๊กอิน (Plugins) ให้เลือกใช้มากมาย เพื่ออำนวยความสะดวกในการพัฒนาไปป์ไลน์ เพื่อทำการสร้างและปรับใช้ซอฟต์แวร์ โดย Jenkins สามารถทำงานได้ด้วยตัวเอง (Standalone) ในกระบวนการของมันเองได้ โดยสามารถทำงานแบบไร้เซิร์ฟเวอร์ (Serverless) ในรูปแบบคอนเทนเนอร์ที่ติดตั้งในระบบส่วนตัวของเรา ซึ่งคือเครื่องคอมพิวเตอร์เสมือนที่เราสร้างขึ้น อีกทั้งยังรองรับระบบปฏิบัติการ (Operation System) ทุกประเภท

2.1.1.2 GitHub Actions [11] คือ บริการที่อยู่ภายใน GitHub ที่ช่วยให้การสร้าง การทดสอบ และการปรับใช้ซอฟต์แวร์เป็นไปโดยอัตโนมัติเช่นกัน และมี GitHub Marketplace มากมายที่จะช่วยปรับปรุงและเพิ่มความสะดวกให้แก่ GitHub Workflows ได้ ทั้งนี้ผู้วิจัยได้มีการเปลี่ยนเครื่องมือที่ใช้ในการทดลอง เนื่องจากวัตถุประสงค์ของงานวิจัยคือการตรวจสอบทรัพยากรของ CI/CD ไปป์ไลน์ แต่ GitHub Actions เป็นบริการที่ทำงานบนเซิร์ฟเวอร์ของ GitHub ทำให้ไม่สามารถตรวจสอบทรัพยากรของกระบวนการ CI/CD ไปป์ไลน์ได้โดยตรง การตรวจสอบทรัพยากรจะต้องติดตั้งรันเนอร์ (Runner) บนเซิร์ฟเวอร์เพิ่มเติม ซึ่งจะส่งผลให้มีการใช้ทรัพยากรมากขึ้นไปอีก ด้วยเหตุผลนี้ Jenkins ซึ่งสามารถติดตั้งและตรวจสอบทรัพยากรได้โดยตรงบนเครื่องเสมือน จึงถูกเลือกใช้ในการทดลอง ตามตารางที่ 2-1 ซึ่งเป็นการเปรียบเทียบข้อมูลและการใช้งานระหว่าง Jenkins และ GitHub Actions ดังนี้

ตารางที่ 2-1 การเปรียบเทียบข้อมูลเครื่องมือสร้างและจัดการ CI/CD ไปป์ไลน์

รายการ	Jenkins	GitHub Actions
การอำนวยความสะดวกในการจัดการ CI/CD ไปป์ไลน์	Jenkins Plugins	GitHub Marketplace
ความสามารถในการรวบรวมเครื่องมือรักษาความปลอดภัยในแต่ละขั้นตอน	✓	✓
การตรวจสอบทรัพยากรเวลา (Execution Time)	✓	✓
การตรวจสอบทรัพยากรหน่วยความจำ (Memory usage)	✓	✗
การตรวจสอบทรัพยากรการประมวลผล (CPU usage)	✓	✗

2.1.2 เครื่องมือรักษาความปลอดภัยที่บูรณาการใน CI/CD ไปป์ไลน์ความปลอดภัยแบบหลายชั้น ในการทดลองนี้ ได้บูรณาการเครื่องมือความปลอดภัยจำนวน 4 เครื่องมือ ได้แก่

2.1.2.1 Snyk [12] เป็นซอฟต์แวร์รักษาความปลอดภัยที่บนคลาวด์ (Cloud-based) ที่ทำหน้าที่ระบุช่องโหว่ในไลบรารี่ (Libraries) การอ้างอิง (Dependency) และ คอนเทนเนอร์ พร้อมทั้งมีระบบแก้ไขช่องโหว่อัตโนมัติในรูปแบบของการ Pull Request ของ GitHub และ Version Control ตัวอื่น ๆ นอกจากนี้ Snyk ยังสามารถบูรณาการเข้ากับ CI/CD ไปป์ไลน์ได้อย่างง่ายดาย

2.1.2.2 Trivy [13] เป็นซอฟต์แวร์โอเพนซอร์ส (Open source) ที่โฮสต์เอง (Self-Hosted) ที่ทำหน้าที่ตรวจจับช่องโหว่ในไลบรารี่ (Libraries) การอ้างอิง (Dependency) และ คอนเทนเนอร์อิมเมจต่าง ๆ (Container Image) เช่น CentOS Package, Ubuntu หรือ Red Hat โดย Trivy ก็เป็นเครื่องมือที่ใช้งานง่ายและสามารถบูรณาการเข้ากับ CI/CD ไปป์ไลน์ได้เป็นอย่างดี

2.1.2.3 SonarQube [14] เป็นซอฟต์แวร์โอเพนซอร์ส (Open source) แบบโฮสต์เอง (Self-Hosted) ที่ใช้ตรวจสอบคุณภาพโค้ด หรือเรียกได้ว่าเป็นเครื่องมือวิเคราะห์โค้ดแบบสถิต (Static Code Analysis) ที่สามารถตรวจจับช่องโหว่ในโค้ดได้มากกว่า 20 ภาษา นอกจากนี้ SonarQube ยังมีส่วนติดต่อผู้ใช้งานแบบกราฟิก (Graphical User Interface: GUI) ที่ใช้งานง่าย และสามารถติดตั้งบนคอมพิวเตอร์เสมือนได้

2.1.2.4 OWASP ZAP [15] เป็นซอฟต์แวร์โอเพนซอร์ส (Open source) ทำหน้าที่ทดสอบความปลอดภัยของเว็บแอปพลิเคชัน หรือเป็นการทดสอบความปลอดภัยแบบไดนามิก (Dynamic Application Security Testing: DAST scan) ที่เป็นการจำลองเจาะระบบเพื่อค้นหาช่องโหว่ในเว็บแอปพลิเคชัน โดย OWASP ZAP สามารถตรวจหาช่องโหว่ได้หลากหลาย เช่น SQL Injection และ cross-site scripting (XSS) หลังจากทดสอบจะมีรายงานผลการทดสอบเพื่อนำไปแก้ไขก่อนที่จะส่งมอบซอฟต์แวร์ไปยังสภาพแวดล้อมการใช้งานจริง จากข้อมูลที่กล่าวมา บ่งชี้ได้ว่า OWASP ZAP เป็นเครื่องมือที่ใช้ในการทดสอบความปลอดภัยในสภาพแวดล้อมการพัฒนา เพื่อระบุและทำการแก้ไขช่องโหว่ก่อนที่จะนำซอฟต์แวร์ไปใช้งานจริงในสภาพแวดล้อมการผลิต ซึ่งจะช่วยเพิ่มความมั่นใจในความปลอดภัยของระบบก่อนที่จะส่งมอบผลิตภัณฑ์ซอฟต์แวร์ให้ลูกค้า

2.1.3 เครื่องมือตรวจสอบทรัพยากร

2.1.3.1 Portainer [16] เป็นซอฟต์แวร์โอเพนซอร์ส (Open source) แบบโฮสต์เอง (Self-Hosted) ที่ใช้จัดการ Container บนเซิร์ฟเวอร์ ซึ่งสามารถกดเริ่ม หยุด ลบ หรือแสดงการบันทึก (Log) ของ Container ตามที่ต้องการได้ โดยการจัดการทั้งหมดนั้น สามารถจัดการได้ผ่านส่วนติดต่อผู้ใช้งานแบบกราฟิก (Graphical User Interface: GUI) ที่ใช้งานง่าย อีกทั้ง Portainer จะแสดงหน้าการแสดงผล (Dashboard) ของทรัพยากรแบบเรียลไทม์ (Real-Time) ประกอบด้วยประเภทของทรัพยากรต่าง ๆ ดังนี้

1) CPU Usage (%) ตรวจสอบว่า Container ที่ต้องการตรวจสอบใช้การประมวลผลมากน้อยเพียงใด

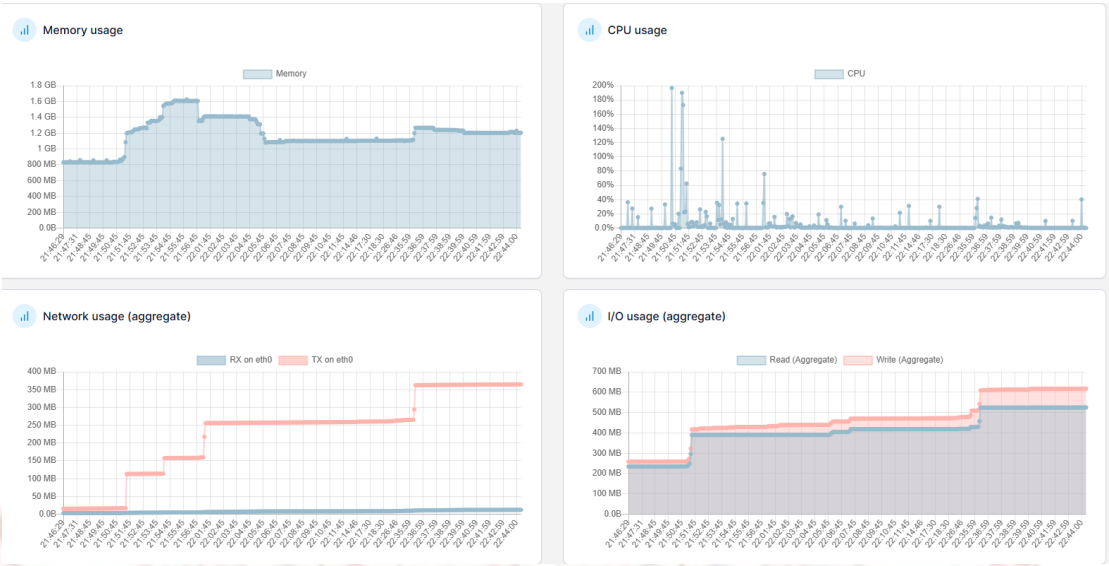
2) Memory Usage (MB) ตรวจสอบการใช้หน่วยความจำของ Container ที่ต้องการตรวจสอบว่าใช้พื้นที่หน่วยความไปเท่าใด

3) Network Traffic (Rx/Tx) ตรวจสอบปริมาณข้อมูลที่รับส่งผ่านเครือข่าย (Network)

4) Block I/O ตรวจสอบการอ่าน/เขียนข้อมูลของคอนเทนเนอร์กับดิสก์

ซึ่งการแสดงผลการใช้ทรัพยากรดังกล่าวของ Portainer จะแสดงผลควบคู่ไปกับข้อมูลเวลาเพื่อให้เห็นภาพรวมของการใช้ทรัพยากรในช่วงเวลาต่างๆ ได้อย่างชัดเจน

การใช้ Portainer ในการทดลองนี้ ช่วยให้สามารถติดตามและวิเคราะห์การใช้ทรัพยากรของคอนเทนเนอร์ในแต่ละขั้นตอนของไปป์ไลน์ CI/CD ได้อย่างละเอียด ทำให้สามารถระบุช่วงเวลาที่มีการใช้ทรัพยากรสูงสุด (Peak Load) และวิเคราะห์ผลกระทบของการเพิ่มเครื่องมือรักษาความปลอดภัยต่อการใช้ทรัพยากรได้ นอกจากนี้ Portainer ยังมีการติดตั้งที่ง่ายดายและใช้ทรัพยากรบนเซิร์ฟเวอร์ไม่มาก โดยภาพที่ 2-1 ที่แสดงตัวอย่างหน้าการแสดงผลของ Portainer



ภาพที่ 2-1 ตัวอย่างหน้าการแสดงผลของ Portainer

2.1.4 เครื่องมืออื่น ๆ ที่จำเป็นในการทดลอง

2.1.4.1 Docker [17] เป็นแพลตฟอร์มโอเพนซอร์ส (Open source) ที่ใช้ในการรันแอปพลิเคชันให้ง่ายต่อการพัฒนาและจัดการ โดยแอปพลิเคชันที่สร้างด้วย Docker จะถูกรวมไว้ในรูปแบบมาตรฐานที่เรียกว่า Container โดยจะรันแยกตัวบนเคอร์เนล (Kernel) ของระบบปฏิบัติการ (Operation System) ทำให้การจัดการมีประสิทธิภาพและง่ายดายยิ่งขึ้น

2.1.4.2 Docker Image คือไฟล์ที่ได้จากการ Build ซึ่งรวบรวมโค้ด ไลบรารี่ และ Dependency ต่าง ๆ เพื่อเตรียมพร้อมสำหรับการปรับใช้

2.1.4.3 Docker Hub เป็นแหล่งจัดเก็บ Docker Images หรือสามารถเรียกได้ว่า Docker Registries ที่สามารถพุช (Push) หรือ พูล (Pull) อิมเมจได้

2.2 งานวิจัยที่เกี่ยวข้อง

2.2.1 ช่องโหว่ในสภาพแวดล้อม CI/CD ไปป์ไลน์

ช่องโหว่ในสภาพแวดล้อม CI/CD ไปป์ไลน์ คือ จุดอ่อนหรือข้อบกพร่องในกระบวนการพัฒนาและส่งมอบซอฟต์แวร์อัตโนมัติ ซึ่งเป็นจุดที่อาจทำให้ผู้โจมตีสามารถเข้าถึง หรือเข้าไปควบคุม เปลี่ยนแปลงซอฟต์แวร์หรือข้อมูลที่สำคัญ ส่งผลให้เกิดภัยคุกคามได้ โดยช่องโหว่นี้อาจเกิดขึ้นได้ในหลาย ๆ ส่วนของไปป์ไลน์ ยกตัวอย่างเช่น การควบคุมการเข้าถึงที่ไม่เหมาะสม ซึ่งอาจทำให้ผู้โจมตีสามารถเข้าถึงข้อมูลลับหรือโค้ดสำคัญได้ และในส่วนของการพัฒนาโค้ด การใช้ Dependencies จากภายนอกเป็นเรื่องปกติสำหรับการพัฒนาโค้ด อย่างไรก็ตาม Dependencies เหล่านั้นอาจมีช่องโหว่หรือทีมพัฒนาอาจสร้างโค้ดที่มีช่องโหว่ด้วยตนเอง [19]

2.2.2 ภัยคุกคามในสภาพแวดล้อม CI/CD ไปป์ไลน์

ภัยคุกคามใน CI/CD ไปป์ไลน์ คือ ความเสี่ยงและช่องโหว่ที่อาจเกิดขึ้นในกระบวนการพัฒนาและส่งมอบซอฟต์แวร์อัตโนมัติและส่งผลเสียต่อความลับ (Confidentiality) ความสมบูรณ์ (Integrity) และความพร้อมใช้งาน (Availability) ตลอดทั้งวงจรของการส่งมอบซอฟต์แวร์ขององค์กร จากการยกตัวอย่างของงานวิจัย [7] เช่น การแทรกโค้ดที่เป็นอันตราย (Code Injection) การใช้การอ้างอิงจากบุคคลที่สามที่เป็นอันตราย (Third-party Malicious Dependencies) หรือ ขาดการทดสอบที่ครอบคลุม (Inadequate Security Testing) ตัวอย่างเช่น หากผู้พัฒนาใช้การอ้างอิงที่ล้าสมัยและมีช่องโหว่ ผู้โจมตีก็อาจใช้ช่องโหว่นั้นในการแทรกโค้ดที่เป็นอันตรายเข้าสู่แอปพลิเคชันของเราได้ ดังนั้น การอัปเดต (Update) Dependency ให้เป็นเวอร์ชันล่าสุด จึงเป็นแนวทางหนึ่งในการป้องกันภัยคุกคามนี้ ด้วยเหตุผลนี้ จากบล็อกของ AWS [18] ได้แนะนำว่าควรบูรณาการมาตรการความปลอดภัยกับทุกขั้นตอนของ CI/CD ไปป์ไลน์ ตั้งแต่การพัฒนา การทดสอบ ไปจนถึงการปรับใช้

2.2.3 CI/CD ไปป์ไลน์ที่มีความปลอดภัย (Secure CI/CD Pipeline)

2.2.3.1 ความหมายของ CI/CD ไปป์ไลน์ที่มีความปลอดภัย

CI/CD ย่อมาจาก Continuous Integration และ Continuous Delivery เป็นหนึ่งในแนวทางปฏิบัติหลักของ DevOps ที่เป็นชุดขั้นตอนอัตโนมัติในการพัฒนาซอฟต์แวร์ เพื่อเผยแพร่โค้ดหรือส่งมอบซอฟต์แวร์ให้กับลูกค้าได้อย่างรวดเร็วและมีประสิทธิภาพมากขึ้น นอกจากนี้ ตาม E-book ของ CircleCI [19] ได้ระบุไว้ว่า CI/CD ไปป์ไลน์ ยังสามารถมีบทบาทสำคัญในการจัดการช่องโหว่ โดยทำการบูรณาการมาตรการและเครื่องมือรักษาความปลอดภัยเข้าไป ทำให้เป็น CI/CD ไปป์ไลน์ที่มีความปลอดภัย

2.2.3.2 ความสำคัญของ CI/CD ไปป์ไลน์ที่มีความปลอดภัย

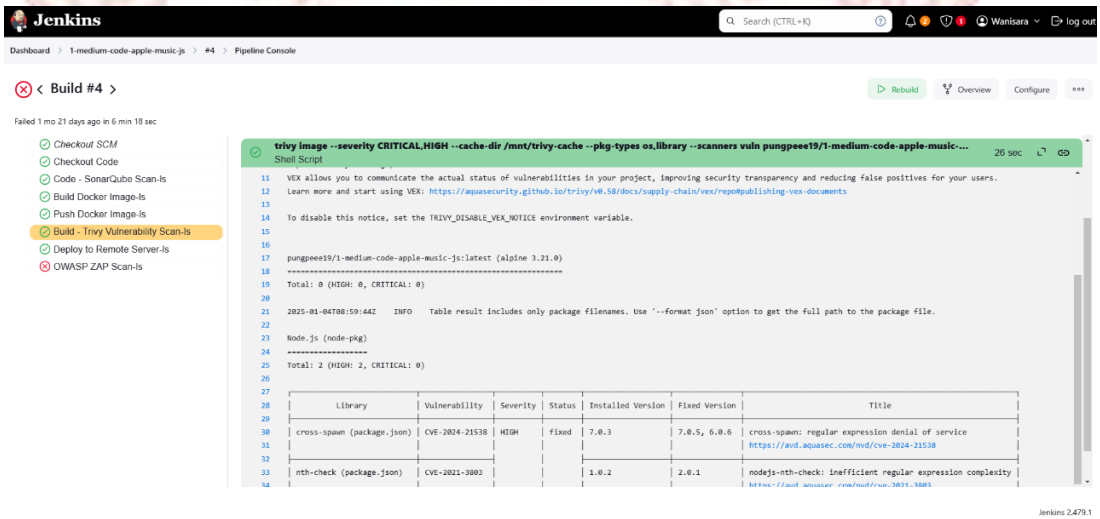
เมื่อได้รวบรวมมาตรการความปลอดภัยที่เหมาะสมแล้ว การผสานเข้าไปในแต่ละขั้นตอนของ CI/CD ไปป์ไลน์จะเป็นเรื่องที่ง่ายดายมากขึ้น จากงานวิจัย [9] ได้นำเสนอว่า ปัจจุบันมีเครื่องมือความปลอดภัยมากมายที่สามารถบูรณาการกับ CI/CD ไปป์ไลน์ได้ ยกตัวอย่างเช่น SonarQube, Clair, Veracode สำหรับการสแกนโค้ดและตรวจจับช่องโหว่ Snyk สำหรับระบุและจัดการช่องโหว่ Trivy สำหรับการตรวจจับ Image และ OWASP ZAP สำหรับทดสอบเจาะระบบเพื่อทดสอบความปลอดภัย เครื่องมือเหล่านี้สามารถลดความเสี่ยงในการนำช่องโหว่เข้าสู่สภาพแวดล้อมการผลิต (Production Environment) ได้ ทำให้ซอฟต์แวร์ที่เผยแพร่มีความปลอดภัย นอกจากนี้ CI/CD ไปป์ไลน์เป็นผลลัพธ์ของการทำงานร่วมกันของการพัฒนาซอฟต์แวร์หลาย ๆ กระบวนการ ประกอบด้วย การพัฒนาโค้ด การทดสอบโค้ด จนไปถึงการส่งมอบซอฟต์แวร์ ดังนั้น การบูรณาการความปลอดภัยเข้ากับ CI/CD ไปป์ไลน์ จะทำให้มีความปลอดภัยอย่างต่อเนื่องและครอบคลุมในระยะยาว ไม่ใช่แค่การลดช่องโหว่เพียงครั้งเดียวแล้วเสร็จสิ้น [19]

2.2.4 ไปป์ไลน์ CI/CD ที่ปลอดภัยในขั้นตอนการสร้าง (Secure Build Stage CI/CD Pipeline)

งานวิจัย [7] ได้เสนอการบูรณาการมาตรการและเครื่องมือรักษาความปลอดภัยในแต่ละขั้นตอนของ CI/CD ไปป์ไลน์ และสามารถลดความเสี่ยงได้จริง อีกทั้งข้อมูล [8] ได้ระบุว่า Build Stage เป็นขั้นตอนที่เกี่ยวข้องกับ DevOps และเป็นจุดรวมของโค้ด Dependencies และ Images เพื่อเตรียมพร้อมสำหรับการ Deploy หากได้ทำการตรวจพบและแก้ไขช่องโหว่ในขั้นตอนนี้ ความเสี่ยงด้านความปลอดภัยจะลดลงก่อนถึงสภาพแวดล้อมการผลิต

2.2.5 เครื่องมือรักษาความปลอดภัยสำหรับขั้นตอนการสร้าง (Build Stage)

งานวิจัย [6, 20] พบว่า Snyk เป็นเครื่องมือรักษาความปลอดภัยที่เกี่ยวข้องกับ Build Stage ซึ่งสามารถตรวจจับช่องโหว่และมีคุณสมบัติในการแก้ไขช่องโหว่โดยอัตโนมัติ จึงช่วยลดช่องโหว่ได้อย่างมีนัยสำคัญ ยิ่งไปกว่านั้น งานวิจัย [12] ได้นำเสนอถึงคุณสมบัติ (Feature) ของ Snyk ที่ช่วยให้กระบวนการรวดเร็วและประหยัดเวลา โดย Snyk จะระบุ Dependencies ที่มีช่องโหว่หรือล้าสมัย พร้อมแนะนำเวอร์ชันใหม่และสร้าง Pull Request บน GitHub โดยอัตโนมัติ เพื่ออัปเดต Dependencies นอกจากนี้ งานวิจัย [13] ได้นำเสนอ Trivy ซึ่งเป็นเครื่องมือตรวจจับช่องโหว่ใน Build Stage โดยเฉพาะ ซึ่งจะสแกน Images, Dependencies และ Libraries ที่เป็นไฟล์ที่เกี่ยวข้องในขั้นตอนนี้ทั้งหมด อีกทั้ง Trivy สามารถบูรณาการเข้ากับ CI/CD ไปป์ไลน์ได้อย่างง่ายดาย และแสดงผลรายงานความปลอดภัยในรูปแบบที่เข้าใจง่าย ซึ่งอยู่ในบันทึกการ Deploy (Deploy Log) ดังภาพที่ 2-2 ที่จะแสดงตัวอย่างผลการรายงานความปลอดภัยของ Trivy ทำให้ทั้งสองเครื่องมือนี้ถูกเลือกใช้ในการทดลองนี้ อย่างไรก็ตาม จากข้อมูล [21] SonarQube สามารถตรวจจับช่องโหว่จากไฟล์ที่เกี่ยวข้องกับ Build Stage ได้เช่นกัน เพียงแต่จะเน้นไปที่การตรวจสอบมาตรฐานโค้ดมากกว่า ดังนั้น SonarQube จึงถูกเลือกมาใช้ในขั้นตอน Code แทน



ภาพที่ 2-2 ตัวอย่างผลรายงานความปลอดภัยของ Trivy ใน Deploy Log บน Jenkins

2.2.6 การตรวจสอบการใช้ทรัพยากร (Resource Usage Monitoring)

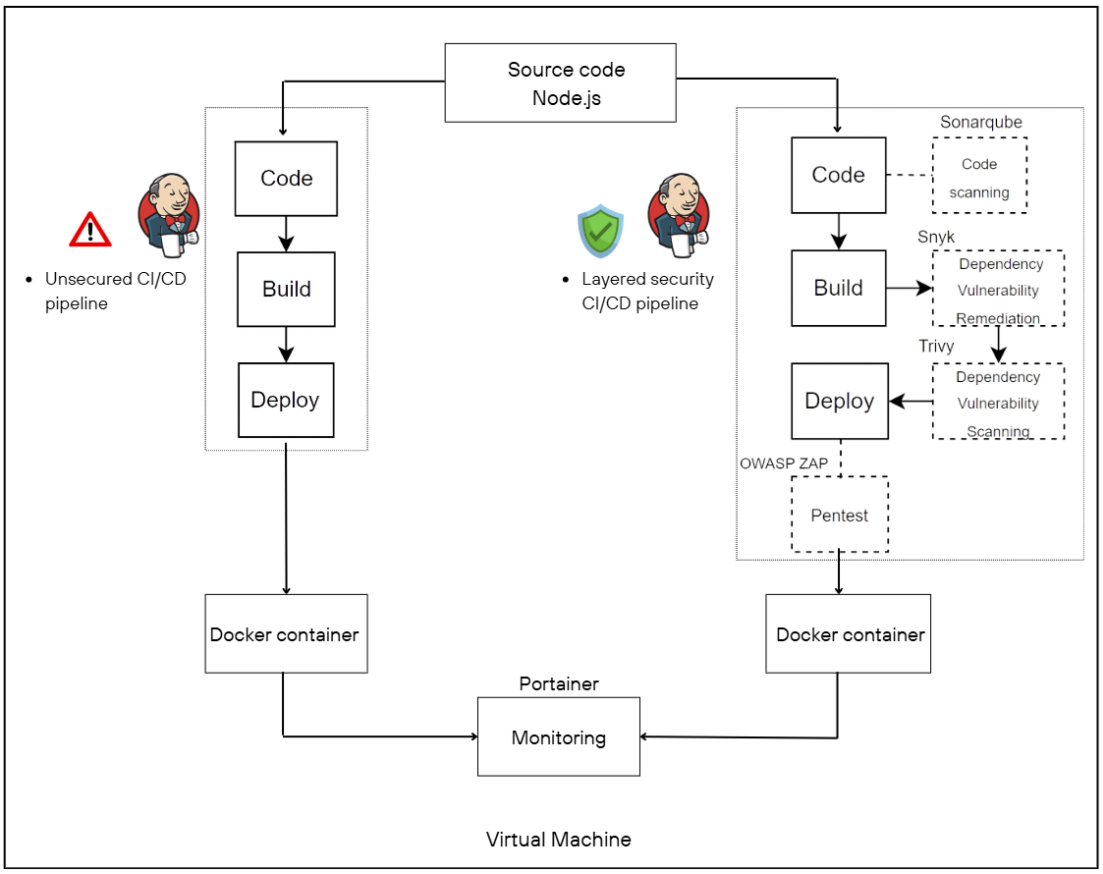
บล็อกของ InfluxData [22] ชี้ให้เห็นว่าการตรวจสอบ CI/CD ไปป์ไลน์อย่างต่อเนื่องเป็นสิ่งสำคัญสำหรับการประเมินประสิทธิภาพของกระบวนการ ในการทดลองนี้ ใช้ Portainer ซึ่งเป็นเครื่องมือจัดการและตรวจสอบคอนเทนเนอร์ เพื่อติดตามการใช้ทรัพยากรหน่วยความจำ และ CPU แบบเรียลไทม์ (Real Time) ในแต่ละขั้นตอนของ CI/CD ไปป์ไลน์ที่มีและไม่มีความปลอดภัย โดยวิเคราะห์แนวโน้มการใช้งานผ่านกราฟแสดงผล (Dashboard) เพื่อให้องค์กรสามารถประเมินผลกระทบของการนำมาตรการความปลอดภัยไปใช้

บทที่ 3
วิธีการดำเนินการวิจัย

ในบทนี้ได้กล่าวถึงภาพรวมการทำงานและวิธีการดำเนินการวิจัย โดยเนื้อหาหลักมีสามหัวข้อ ได้แก่ ภาพรวมการทำงาน เครื่องมือที่ใช้ และขั้นตอนการดำเนินงานวิจัย มีรายละเอียดดังต่อไปนี้

3.1 ภาพรวมการทำงานวิจัย

งานวิจัยนี้ทำการเปรียบเทียบ CI/CD ไปป์ไลน์สองรูปแบบ ได้แก่ CI/CD ไปป์ไลน์ที่ไม่มีความปลอดภัยและไปป์ไลน์ที่มีความปลอดภัยแบบหลายชั้น โดยเน้นที่การลดช่องโหว่ใน Build Stage ผ่านการดำเนินการด้วย Jenkins พร้อมตรวจสอบเวลาดำเนินการของไปป์ไลน์ และการใช้ทรัพยากรของเครื่องคอมพิวเตอร์เสมือนได้แก่ หน่วยความจำ และ CPU ด้วย Portainer รายละเอียดแสดงในภาพที่ 3-1



ภาพที่ 3-1 ภาพรวมการทำงานวิจัย

จากภาพที่ 3-1 สามารถอธิบายรายละเอียดกระบวนการทำงานของงานวิจัยได้ดังต่อไปนี้

1) Source code (ด้านบนของภาพ)

ในการทดลองนี้ ได้เตรียมไฟล์โค้ด Node.js จำนวน 9 ไฟล์ แบ่งเป็น 3 ขนาด ได้แก่ขนาดเล็ก กลาง และใหญ่ ขนาดละ 3 ไฟล์ และเป็นรูปแบบโค้ดที่แตกต่างกันไป เช่น หน้าเว็บ หรือ เว็บแอปพลิคชัน (Web Application) เพื่อใช้เป็นข้อมูลนำเข้า (Input) สำหรับการปรับใช้ (Deploy) ผ่านไปป์ไลน์ทั้งสองรูปแบบ

2) Unsecured CI/CD Pipeline หรือ CI/CD ไปป์ไลน์ที่ไม่มีความปลอดภัย (ฝั่งซ้ายของภาพ)

ไปป์ไลน์นี้ทำงานโดยใช้ Jenkins เป็นเครื่องมือหลักในดำเนินการตาม 3 ขั้นตอน ได้แก่

- Code Stage หรือขั้นตอนการเตรียมโค้ด เพื่อรับ Source code ที่พัฒนาโดย Node.js ที่ไม่มีการตรวจสอบด้านความปลอดภัย
- Build Stage หรือขั้นตอนการสร้าง เพื่อสร้าง Docker Image ของแอปพลิเคชันโดยไม่มีการสแกนช่องโหว่ของ Dependency
- Deploy Stage หรือขั้นตอนการปรับใช้ เพื่อดำเนินการ Deploy ไปยัง Docker Container บนเครื่องคอมพิวเตอร์เสมือน (Virtual Machine)

เนื่องจากไปป์ไลน์นี้ไม่มีมาตรการด้านความปลอดภัยใด ๆ จึงมีความเสี่ยงสูงที่โค้ดที่มีช่องโหว่จะถูกนำไปใช้งานจริง

3) Layered secure CI/CD Pipeline หรือ CI/CD ไปป์ไลน์ที่มีความปลอดภัยแบบหลายชั้น (ฝั่งของของภาพ)

ไปป์ไลน์นี้ได้รับการปรับปรุงโดยเพิ่มขั้นตอนที่เป็นเครื่องมือรักษาความปลอดภัย เพื่อทำการลดช่องโหว่รวมถึงป้องกันช่องโหว่ที่อาจเกิดขึ้นในกระบวนการพัฒนา โดยมีขั้นตอนดังต่อไปนี้

- Code Stage หรือขั้นตอนการเตรียมโค้ด เพื่อรับ Source code ที่พัฒนาโดย Node.js และทำการสแกนความปลอดภัยของโค้ดผ่าน SonarQube เพื่อตรวจสอบคุณภาพของโค้ดให้ตรงตามมาตรฐานและระบุช่องโหว่ของโค้ด เพื่อให้ทางทีมพัฒนาโค้ดได้นำไปปรับปรุงแก้ไข
- Build Stage หรือขั้นตอนการสร้างซอฟต์แวร์ หลังจากสร้าง Docker Image จะใช้ Snyk เพื่อแก้ไขช่องโหว่ของ Dependency หาก Snyk พบช่องโหว่จะทำการแก้ไขช่องโหว่นั้นอัตโนมัติตามเงื่อนไขที่ผู้วิจัยได้ระบุในการทดลองนี้ และใช้ Trivy เพื่อตรวจจับจำนวนช่องโหว่นั้นอีกครั้ง
- Deploy Stage หรือขั้นตอนการปรับใช้ เพื่อดำเนินการ Deploy ไปยัง Docker Container บนเครื่องคอมพิวเตอร์เสมือน

- Test Stage หรือขั้นตอนการทดสอบ ที่มี OWASP ZAP เป็นเครื่องมือทดสอบความปลอดภัยของเว็บแอปพลิเคชันหรือทำการทดสอบเจาะระบบหลังจากการ Deploy เสร็จสิ้น เพื่อตรวจสอบช่องโหว่ก่อนนำไปใช้งานจริง

ไปป์ไลน์นี้ออกแบบมาเพื่อให้มีความปลอดภัยมากขึ้น ลดความเสี่ยงจากช่องโหว่ในแต่ละขั้นตอน แก้ไขช่องโหว่ใน Build Stage และเพิ่มความมั่นใจว่าโค้ดที่ Deploy จะปลอดภัย

4) Monitoring (ด้านล่างของภาพ)

การทดลองนี้ได้ดำเนินการทดสอบการ Deploy Docker Container บนเครื่องคอมพิวเตอร์ผ่านไปป์ไลน์ CI/CD ทั้งสองรูปแบบ ได้แก่ ไปป์ไลน์ที่มีความปลอดภัยแบบหลายชั้น และไปป์ไลน์ที่ไม่มีความปลอดภัย โดยไปป์ไลน์ทั้งสองรูปแบบถูกกำหนดไว้ใน Jenkinsfile สำหรับโค้ดจำนวน 9 ไฟล์ เพื่อให้ครอบคลุมขนาดของโค้ดที่แตกต่างกัน ประกอบด้วยโค้ดขนาดเล็ก กลาง และใหญ่ โดยการทดลองนี้ ไปป์ไลน์แต่ละรูปแบบถูกรันทั้งหมดไม่ต่ำกว่า 18 ครั้ง (โค้ดจำนวน 9 ไฟล์ x ไปป์ไลน์ทั้ง 2 รูปแบบ)

ในระหว่างการรันไปป์ไลน์แต่ละครั้ง จะมีการใช้ Portainer ซึ่งเป็นเครื่องมือจัดการ Docker Container เพื่อติดตามแนวโน้มการใช้ทรัพยากรระบบแบบเรียลไทม์ ได้แก่ หน่วยความจำ (Memory) และการประมวลผล (CPU) และจะทำการบันทึกผลการใช้ทรัพยากรดังกล่าวรวมไปถึงเวลาดำเนินการของไปป์ไลน์แต่ละครั้งผ่าน Jenkins เพื่อนำมาวิเคราะห์และเปรียบเทียบผลลัพธ์ระหว่างไปป์ไลน์ทั้งสองรูปแบบ

3.2 เครื่องมือที่ใช้ในการวิจัย

3.2.1 สภาพแวดล้อมการพัฒนาที่โฮสต์เอง

3.2.1.1 คอมพิวเตอร์เสมือนบนไมโครซอฟต์ อาชัวร์ (Microsoft Azure)

- หน่วยการประมวลผล (CPU) 2 vCPUs
- หน่วยความจำ (Memory) 8 กิกะไบต์ (RAM 8 GB)
- ระบบปฏิบัติการลินุกซ์ (Linux) (Ubuntu 24.04 LTS)

3.2.1.2 Docker ระบบจัดการคอนเทนเนอร์

3.2.1.3 Jenkins เพื่อใช้ในการรัน CI/CD ไปป์ไลน์ทั้งสองรูปแบบ

3.2.1.4 Portainer เพื่อใช้ในการตรวจสอบทรัพยากรของคอนเทนเนอร์ Jenkins ในระหว่างกระบวนการ CI/CD

3.2.2 สภาพแวดล้อมการพัฒนาที่ใช้งานจากภายนอก

3.2.2.1 Docker Hub เพื่อจัดเก็บ Docker Image ที่สร้างเสร็จเรียบร้อย โดยกำหนดค่าเป็น Container registry แบบส่วนตัว (Private)

3.2.2.2 GitHub เพื่อจัดเก็บโค้ดที่จะถูกนำมาปรับใช้ผ่าน CI/CD ไปป์ไลน์ทั้งสองรูปแบบ

3.2.2.3 Visual Studio Code สำหรับการพัฒนา Jenkins CI/CD เวิร์คโฟลว์

3.3 ขั้นตอนการดำเนินงานวิจัย

หลังจากสร้างคอมพิวเตอร์เสมือนและติดตั้งคอนเทนเนอร์ต่าง ๆ ที่เป็นสภาพแวดล้อมที่จำเป็นในการพัฒนาแล้ว ขั้นตอนการดำเนินงานและพัฒนาระบบมีรายละเอียดดังต่อไปนี้

3.3.1 การเตรียมโค้ด (Source code Input)

โค้ดที่ใช้เป็น Input มาจากบริษัทซอฟต์แวร์ ที่พัฒนาด้วย Node.js โดย Node.js เป็นแพลตฟอร์ม (Platform) ที่ใช้รันในภาษาจาวาสคริปต์ (JavaScript) ที่ได้รับความนิยมสูงสุดในการพัฒนา ในการทดลองนี้ ได้เลือกโค้ดและระบุให้เป็น 3 ขนาด ได้แก่ เล็ก กลาง และใหญ่ โดยมีรูปแบบที่แตกต่างกัน เช่น Front-end, Back-end และเว็บแอปพลิเคชัน ดังตารางที่ 3-1 ซึ่งแสดงการเปรียบเทียบขนาดและคุณลักษณะของโค้ด โดยมีรายละเอียด ได้แก่ ขนาดของโค้ด จำนวนบรรทัดของโค้ด (Lines of Code: LOC) ตัวอย่างรูปแบบของโค้ด และลักษณะเฉพาะ (Characteristics) ทั้งนี้เพื่อทำให้มั่นใจว่า CI/CD ไปป์ไลน์ที่สร้างขึ้นทั้งสองรูปแบบนั้น มีความยืดหยุ่นและสามารถปรับใช้ได้กับโค้ดทุกรูปแบบ

ตารางที่ 3-1 การเปรียบเทียบขนาดและคุณลักษณะของโค้ด

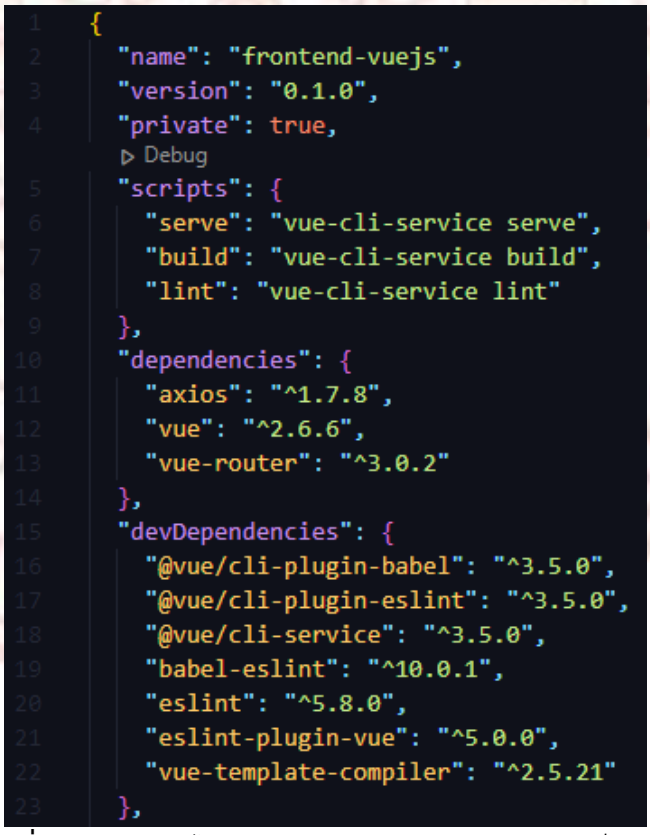
ขนาด	จำนวนบรรทัดของโค้ด (x)	ตัวอย่างรูปแบบของโค้ด	ลักษณะเฉพาะ
เล็ก	$x < 5000$ บรรทัด	Front-end	จำนวน Dependencies และ Libraries น้อย มีหน้าเว็บเรียบง่าย โค้ดไม่ซับซ้อน และอิมเมจมีขนาดเล็ก
กลาง	$5000 < x \leq 9000$ บรรทัด	Back-end และเว็บแอปพลิเคชัน	จำนวน Dependencies และ Libraries ปานกลาง
ใหญ่	$9,000 < x \leq 20,000$ บรรทัด	Front-end และเว็บแอปพลิเคชัน	จำนวน Dependencies และ Libraries มาก

รายละเอียดจำนวน Dependencies ที่โค้ดแต่ละขนาดใช้ ดังตารางที่ 3-2 โดย Dependencies คือ ไลบรารีหรือโมดูลที่จำเป็นสำหรับการพัฒนาโค้ด ในขณะที่ DevDependencies จะใช้เฉพาะใน

สภาพแวดล้อมการพัฒนาเท่านั้น โดยภาพที่ 3-2 แสดงตัวอย่างไฟล์ package.json ของโค้ดขนาดเล็ก ภาพที่ 3-3 แสดงตัวอย่างไฟล์ package.json ของโค้ดขนาดกลาง และภาพที่ 3-4 แสดงตัวอย่างไฟล์ package.json ของโค้ดขนาดใหญ่ ซึ่งเป็นไฟล์ที่ระบุรายการ Dependencies ทั้งหมด

ตารางที่ 3-2 รายละเอียดจำนวนของ Dependencies ที่โค้ดแต่ละขนาดใช้

ขนาด	จำนวน Dependencies	จำนวน DevDependencies	รวม
เล็ก	3	7	10
กลาง	10	1	11
ใหญ่	19	6	25



```
{
  "name": "frontend-vuejs",
  "version": "0.1.0",
  "private": true,
  ▷ Debug
  "scripts": {
    "serve": "vue-cli-service serve",
    "build": "vue-cli-service build",
    "lint": "vue-cli-service lint"
  },
  "dependencies": {
    "axios": "^1.7.8",
    "vue": "^2.6.6",
    "vue-router": "^3.0.2"
  },
  "devDependencies": {
    "@vue/cli-plugin-babel": "^3.5.0",
    "@vue/cli-plugin-eslint": "^3.5.0",
    "@vue/cli-service": "^3.5.0",
    "babel-eslint": "^10.0.1",
    "eslint": "^5.8.0",
    "eslint-plugin-vue": "^5.0.0",
    "vue-template-compiler": "^2.5.21"
  },
```

ภาพที่ 3-2 ตัวอย่างไฟล์ package.json ของโค้ดขนาดเล็ก

จากภาพ package.json เป็นไฟล์ที่ระบุ Dependencies ของที่โปรเจกต์ Node.js จำเป็นต้องใช้ ได้แก่ Dependencies และ DevDependencies อีกทั้งไฟล์นี้ได้ระบุเวอร์ชันของ Dependencies ที่

เป็น Semantic Versioning (SemVer) ซึ่งเป็นรูปแบบการจัดเวอร์ชันที่ได้รับความนิยม เพื่อกำหนดเวอร์ชันของแพ็กเกจ ทำให้สามารถควบคุมการอัปเดตได้อย่างมีประสิทธิภาพ โดยไฟล์นี้มีโครงสร้างแบบเจซัน (JSON) และประกอบด้วยข้อมูลสำคัญเกี่ยวกับโปรเจกต์ เช่น ชื่อ (name) เวอร์ชัน (version) และรายการ Dependencies ที่โปรเจกต์ใช้

การใช้ Dependencies ที่ระบุในไฟล์ package.json ช่วยลดเวลาในการพัฒนาโค้ด เนื่องจากสามารถใช้ชุดโค้ดสำเร็จรูปหรือแพ็กเกจ (Package) ได้ ทำให้ผู้พัฒนาไม่จำเป็นต้องเขียนโปรแกรมเองทั้งหมด นอกจากนี้ เวอร์ชันของ Dependencies ที่ระบุในไฟล์ package.json ยังสามารถควบคุมการติดตั้ง Dependencies ให้เป็นไปตามเวอร์ชันที่กำหนด ซึ่งช่วยลดปัญหาความขัดแย้งระหว่าง Dependencies และเพิ่มเสถียรภาพของโค้ด

```
{} package.json > ...
{
  "name": "apple-music-js",
  "version": "0.1.0",
  "private": true,
  "homepage": "http://tannerv.com/music",
  "dependencies": {
    "feather-icons": "^4.7.3",
    "react": "^16.5.0",
    "react-dom": "^16.12.0",
    "react-lazy-load": "^3.0.13",
    "react-rangeslider": "^2.2.0",
    "react-redux": "^5.0.7",
    "react-scripts": "^5.0.0",
    "redux": "^4.0.0",
    "redux-thunk": "^2.3.0",
    "styled-components": "^5.0.0"
  },
  "scripts": {
    "start": "react-scripts start",
    "build": "react-scripts build",
    "test": "react-scripts test --env=jsdom",
    "eject": "react-scripts eject"
  },
  "devDependencies": {
    "redux-devtools-extension": "^2.13.5"
  },
  "browserslist": [
    ">0.2%",
    "not dead",
    "not ie <= 11",
    "not op_mini all"
  ]
}
```

ภาพที่ 3-3 ตัวอย่างไฟล์ package.json ของโค้ดขนาดกลาง

จากภาพที่แสดงตัวอย่างไฟล์ package.json ของโค้ดทุกขนาด ซึ่งเป็นหนึ่งในสามขนาดโค้ดที่ใช้ในการทดลองนี้ อีกทั้งแต่ละภาพยังชี้ให้เห็นถึงความแตกต่างของไฟล์ package.json ของโค้ดขนาดเล็ก กลาง และใหญ่ จากการเปรียบเทียบช่วยให้เห็นถึงความแตกต่างของ Dependencies ที่ใช้ในแต่ละขนาดโค้ดซึ่งจะส่งผลกระทบต่อการใช้ทรัพยากรระบบและเวลาดำเนินการของ CI/CD ไปป์ไลน์

```
{
  "name": "vekin",
  "version": "0.1.0",
  "private": true,
  ▷ Debug
  "scripts": {
    "dev": "next dev",
    "build": "next build",
    "start": "next start",
    "lint": "next lint"
  },
  "dependencies": {
    "@emailjs/browser": "^3.10.0",
    "@react-google-maps/api": "^2.18.1",
    "axios": "^1.7.8",
    "clsx": "^2.1.0",
    "google-map-react": "^2.2.0",
    "next": "^14.2.15",
    "next-transpile-modules": "^10.0.1",
    "primereact": "^10.0.2",
    "react": "^18.2.0",
    "react-dom": "^18.2.0",
    "react-google-recaptcha": "^2.1.0",
    "react-orgchart": "^1.0.5",
    "remixicon": "^2.5.0",
    "resend": "^3.1.0",
    "sharp": "^0.33.2",
    "swiper": "^8.4.5",
    "swr": "^2.1.2",
    "tailwind-merge": "^2.2.1",
    "xmlhttprequest": "^1.8.0"
  },
  "devDependencies": {
    "@tailwindcss/typography": "^0.5.9",
    "autoprefixer": "^10.4.14",
    "eslint": "8.45.0",
    "eslint-config-next": "13.4.12",
    "postcss": "^8.4.27",
    "tailwindcss": "^3.4.1"
  }
```

ภาพที่ 3-4 ตัวอย่างไฟล์ package.json ของโค้ดขนาดใหญ่

3.3.2 การพัฒนา CI/CD ไปป์ไลน์ทั้งสองรูปแบบ

ไปป์ไลน์ทั้งสองรูปแบบจะถูกรันด้วย Jenkins ในเครื่องคอมพิวเตอร์เสมือนเครื่องเดียวกัน มีรายละเอียดดังต่อไปนี้

1) CI/CD ไปป์ไลน์ที่ไม่มีความปลอดภัย

CI/CD ไปป์ไลน์ที่ไม่มีความปลอดภัยนั้นประกอบไปด้วยขั้นตอนพื้นฐาน ได้แก่ การตรวจสอบโค้ด การสร้าง และการปรับใช้ ซึ่งเป็นรูปแบบที่ธรรมดา รวดเร็ว และสามารถเผยแพร่โค้ดออกสู่สาธารณะได้ทันที อย่างไรก็ตาม รูปแบบนี้อาจมีความเสี่ยงต่าง ๆ เกิดขึ้นตามมา เนื่องจากไม่มีการบูรณาการเครื่องมือรักษาความปลอดภัยในขั้นตอนใดๆ ของไปป์ไลน์ เช่น อาจถูกผู้โจมตีเจาะระบบแอปพลิเคชันผ่านทางช่องโหว่ที่ค้นพบในโค้ดหรือ Dependencies ที่ใช้ ภาพที่ 3-5 แสดงเวิร์กโฟลว์ CI/CD ที่ไม่มีความปลอดภัย ซึ่งถูกกำหนดไว้ใน Jenkinsfile

```
pipeline {
    agent any

    environment {
        DOCKER_TOKEN = credentials('DOCKER_TOKEN')
        SSH_PRIVATE_KEY = credentials('ssh-private-key')
        USER = credentials('server-user')
        HOST = credentials('server-host')
    }
    stages {
        stage('Checkout Code') {
            steps {
                checkout scm
            }
        }
        stage('Build Docker Image') {
            steps {
                script {
                    docker.build("pungpeee19/1-medium-code-apple-music-js:latest")
                }
            }
        }
        stage('Push Docker Image') {
            steps {
                script {
                    sh '''
                    echo ${DOCKER_TOKEN} | docker login -u pungpeee19 --password-stdin
                    docker push pungpeee19/1-medium-code-apple-music-js:latest
                    '''
                }
            }
        }
        stage('Deploy to Remote Server') {
            steps {
                sshagent(credentials: ['ssh-private-key']) {
                    sh '''
                    ssh -o StrictHostKeyChecking=no ${USER}@${HOST} "
                        echo ${DOCKER_TOKEN} | docker login -u pungpeee19 --password-stdin && \
                        docker pull pungpeee19/1-medium-code-apple-music-js:latest && \
                        docker run -d --restart=always -p 1001:3000 --name 1-medium-code-apple-music-js pungpeee19/1-medium-code-apple-music-js:latest
                    "
                    '''
                }
            }
        }
    }
}
```

ภาพที่ 3-5 Jenkinsfile ของ CI/CD ไปป์ไลน์ที่ไม่มีความปลอดภัย

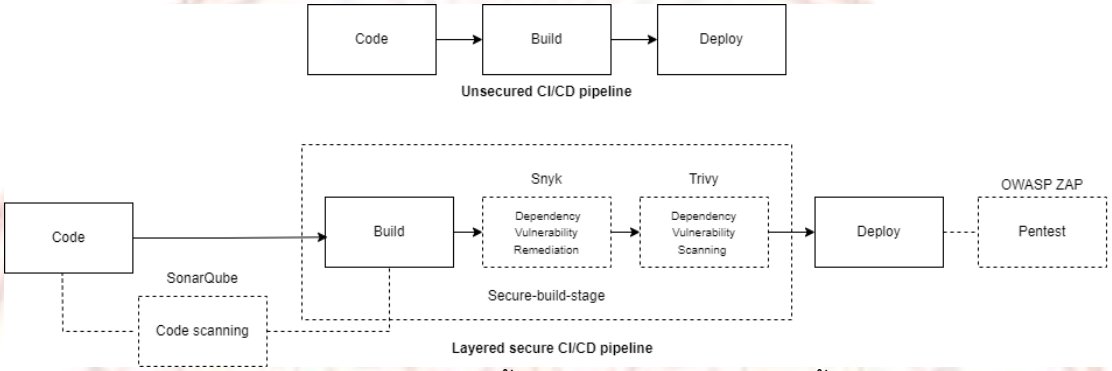
2) CI/CD ไปป์ไลน์ที่มีความปลอดภัยแบบหลายชั้น

CI/CD ไปป์ไลน์ที่มีความปลอดภัย จะผสานรวมเครื่องมือรักษาความปลอดภัยในแต่ละขั้นตอน โดยเพิ่มขั้นตอนจากไปป์ไลน์แบบธรรมดา ดังนี้

- ขั้นตอนการสแกนโค้ดด้วย SonarQube
- ขั้นตอนการแก้ไขช่องโหว่ Dependency ด้วย Snyk
- ขั้นตอนการตรวจจับช่องโหว่ด้วย Trivy

- ขั้นตอนการทดสอบด้วย OWASP ZAP

ด้วยเหตุนี้ จึงถูกเรียกว่า CI/CD ไปป์ไลน์ที่มีความปลอดภัยแบบหลายชั้น เนื่องจากมีการเพิ่มเครื่องมือความปลอดภัยเข้าไปในหลาย ๆ ขั้นตอน ยิ่งไปกว่านั้น ใน Build Stage ของไปป์ไลน์ที่มีความปลอดภัยแบบหลายชั้น จะถูกดำเนินการเพื่อลดช่องโหว่จาก Dependency หรือ Library ต่าง ๆ ที่โค้ดใช้ ดังที่แสดงในภาพที่ 3-6 ซึ่งเปรียบเทียบขั้นตอนของ CI/CD ไปป์ไลน์ทั้งสองรูปแบบ



ภาพที่ 3-6 การเปรียบเทียบขั้นตอนของ CI/CD ไปป์ไลน์ทั้งสองรูปแบบ

3.3.3 การบูรณาการเครื่องมือความปลอดภัยเข้ากับ CI/CD ไปป์ไลน์ที่มีความปลอดภัยแบบหลายชั้น

ในการทดลองนี้ ได้รวบรวมเครื่องมือรักษาความปลอดภัยที่เกี่ยวข้องกับแต่ละขั้นตอนของ CI/CD ไปป์ไลน์ โดยเครื่องมือเหล่านี้ถูกติดตั้งด้วยเหตุผลที่ต่างกัน เพื่อความสะดวกในการใช้งาน โดยแต่ละขั้นตอนใช้เครื่องมือดังต่อไปนี้

3.3.3.1 SonarQube ในขั้นตอนการตรวจสอบโค้ด

SonarQube ถูกติดตั้งบนเครื่องคอมพิวเตอร์เสมือนในรูปแบบคอนเทนเนอร์ ที่แยกต่างหากจากคอนเทนเนอร์ Jenkins โดยมีการกำหนดพอร์ต (Port) ที่แตกต่างกัน แต่ใช้ที่อยู่ไอพี (IP Address) เดียวกัน ซึ่งสามารถเรียกได้ว่าเป็นการใช้งานคอนเทนเนอร์หลายตัวบนโฮสต์เดียว (Multi-Container Deployment on a Single Host) โดยการติดตั้งในรูปแบบนี้ จะแตกต่างจากการติดตั้งเครื่องมือรักษาความปลอดภัยอื่นๆ ที่ใช้ปลั๊กอินของ Jenkins เพื่อความสะดวกในการพัฒนา ซึ่งรายละเอียดการติดตั้ง SonarQube แสดงในภาคผนวก ข โดยใช้คำสั่ง build docker image เพื่อสร้าง Docker Image ของ SonarQube แยกออกมาก่อนและคำสั่ง docker run เพื่อรันคอนเทนเนอร์ SonarQube บนเครื่องคอมพิวเตอร์เสมือน และการกำหนดค่า SonarQube กับโปรเจกต์ที่ต้องการสแกนใน Jenkins Workflows แสดงอยู่ในภาพที่ 3-7 ถึงแม้ว่าการติดตั้งผ่านปลั๊กอินจะสะดวกกว่า แต่ในการทดลองนี้ได้เลือกติดตั้ง SonarQube ในรูปแบบ Multi-Container Deployment on a Single Host

เพื่อให้สอดคล้องกับแนวทางปฏิบัติจริงขององค์กร เนื่องจาก SonarQube มีความเกี่ยวข้องกับทีมพัฒนาโค้ดมากที่สุด การแยกคอนเทนเนอร์ทำให้ทีมพัฒนาและทีม DevOps สามารถจัดการ SonarQube ได้อย่างอิสระและสะดวกยิ่งขึ้น ซึ่งช่วยให้การจัดการมาตรการรักษาความปลอดภัยเป็นไปอย่างมีประสิทธิภาพ

```
stage('Code - SonarQube Scan-ls') {
    steps {
        script {
            docker.image('sonarsource/sonar-scanner-cli:latest').inside {
                sh '''
                sonar-scanner \
                    -Dsonar.projectKey=1-small-code-dr \
                    -Dsonar.sources=. \
                    -Dsonar.host.url=${SONAR_HOST_URL} \
                    -Dsonar.login=${SONAR_TOKEN}
                '''
            }

        }
    }
}
```

ภาพที่ 3-7 การกำหนดค่า SonarQube ในเวิร์คโฟลว์ Jenkinsfile

3.3.3.2 Snyk ในขั้นตอนการสร้าง

Snyk ถูกติดตั้งผ่านปลั๊กอินของ Jenkins บนคอนเทนเนอร์ Jenkins เพื่อประหยัดทรัพยากรของคอมพิวเตอร์เสมือน โดย Build Stage นี้เกี่ยวข้องกับทีม DevOps โดยตรงที่อาจสามารถแก้ไขช่องโหว่ได้ทันที อย่างไรก็ตาม ขั้นตอนนี้ยังมีส่วนเกี่ยวข้องกับทีมพัฒนาโค้ดเช่นกัน ทำให้การสื่อสารระหว่างสองทีมเป็นเรื่องที่สำคัญ ซึ่งควรที่จะร่วมแก้ไขช่องโหว่ร่วมกัน นอกจากนี้ ขั้นตอน Snyk สามารถกำหนดเงื่อนไขต่าง ๆ ได้ตามความต้องการ ดังแสดงในภาพที่ 3-8 ซึ่งเป็นการกำหนดค่า Snyk ในเวิร์คโฟลว์ Jenkinsfile พร้อมระบุเงื่อนไขต่าง ๆ ในการทดลองนี้ได้กำหนดเงื่อนไขในระดับความรุนแรง (Severity Threshold) ให้เป็น “high” ซึ่งหมายถึงให้แสดงผลลัพธ์ของช่องโหว่ที่มีระดับความรุนแรงตั้งแต่ high (ระดับสูง) ขึ้นไป ซึ่งได้แก่ High และ Critical (ระดับวิกฤต) อีกทั้งยังกำหนดเงื่อนไขอีกหนึ่งค่าให้เป็นค่า “true” หมายถึงให้ดำเนินการไปป์ไลน์ต่อไปแม้จะเจอช่องโหว่ก็ตาม ซึ่งจะตรงข้ามกับ “false” ที่จะทำการหยุดไปป์ไลน์ทันทีตั้งแต่ขั้นตอน Snyk เมื่อ Snyk พบช่องโหว่

```
    stage('Build Fix - Snyk Scan dependencies') {
        steps {
            script {
                def snykScan = sh(
                    script: 'snyk test --token=$SNYK_TOKEN --file=package-lock.json
                    --severity-threshold=high || true',
                    returnStatus: true
                )
            }
        }
    }
}
```

ภาพที่ 3-8 การกำหนดค่า Snyk ในเวิร์คโฟลว์ Jenkinsfile

3.3.3.3 Trivy ในขั้นตอนการสร้าง

Trivy ถูกติดตั้งโดยตรงในคอนเทนเนอร์ Jenkins ด้วย Docker in Docker เพื่อให้ตัวควบคุม (Controller) ของ Jenkins สามารถเรียกใช้ Docker Images ในสภาพแวดล้อมเดียวกันได้ เพื่อความรวดเร็วของไปป์ไลน์ และผลรายงานความปลอดภัยของ Trivy อยู่ในบันทึกการ Deploy ซึ่งสะดวกต่อการใช้งานของทีม DevOps โดยการกำหนดค่าใน Jenkins Workflows ดังแสดงในภาพที่ 3-9

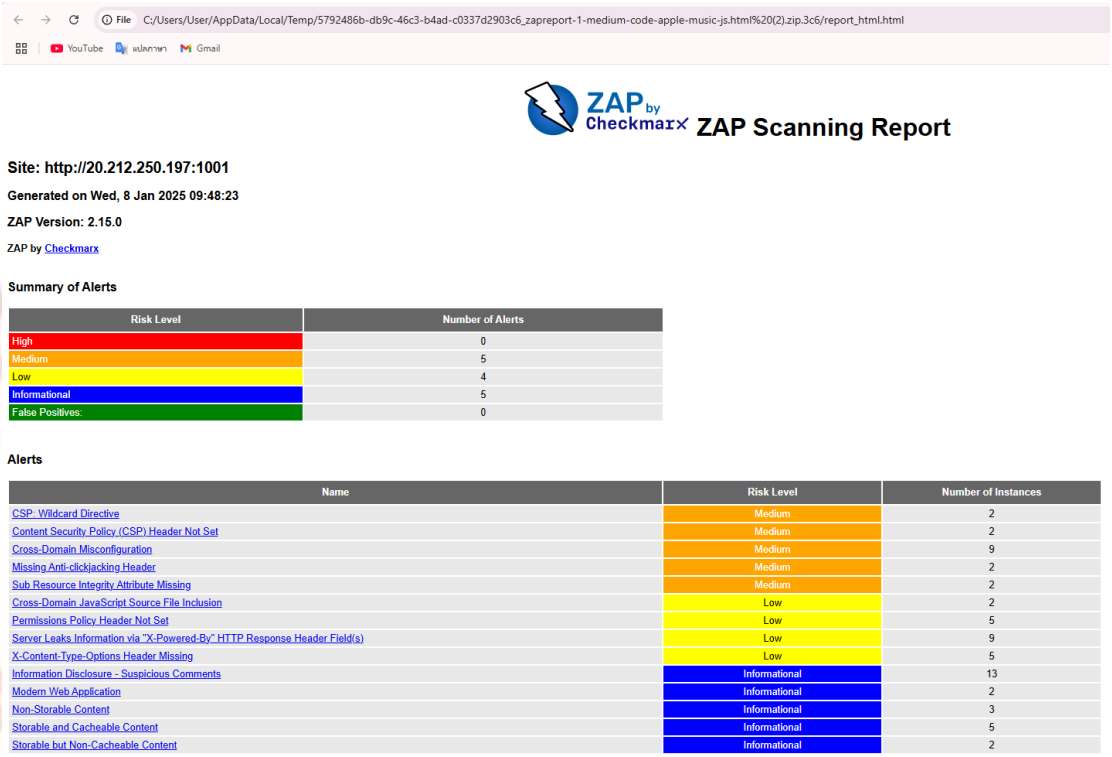
```
stage('Build - Trivy Vulnerability Scan-ls') {
    steps {
        script {
            sh '''
                trivy image --severity CRITICAL,HIGH \
                   --cache-dir /mnt/trivy-cache \
                   --pkg-types os,library \
                   --scanners vuln \
                   pungpeee19/1-medium-code-apple-music-js:latest

            '''
        }
    }
}
```

ภาพที่ 3-9 การกำหนดค่า Trivy ในเวิร์คโฟลว์ Jenkinsfile

3.3.3.4 OWASP ZAP ในขั้นตอนการทดสอบ

OWASP ZAP ถูกติดตั้งในคอนเทนเนอร์ Jenkins ด้วย Docker in Docker เช่นกัน โดยผลรายงานความปลอดภัยของ OWASP ZAP จะอยู่ในรูปของไฟล์ (File) .html ตามภาพที่ 3-10 หรือ .pdf ซึ่งสามารถกำหนดได้ตามความต้องการ โดยถูกจัดเก็บอยู่ในไดเร็กทอรี (Directory) ที่กำหนดเอง

นอกจากนี้ หากเป็นไปตามแนวปฏิบัติจริงขององค์กร จะสามารถทำให้องค์กรส่งรายงานให้กับลูกค้าหรือผู้ที่เกี่ยวข้องก่อนการใช้งานในสภาพแวดล้อมการผลิต กำหนดค่าเพื่อบูรณาการกับ Jenkins Workflows ดังแสดงในภาพที่ 3-11



ZAP by Checkmarx ZAP Scanning Report

Site: http://20.212.250.197:1001

Generated on Wed, 8 Jan 2025 09:48:23

ZAP Version: 2.15.0

ZAP by Checkmarx

Summary of Alerts

Risk Level	Number of Alerts
High	0
Medium	5
Low	4
Informational	5
False Positives:	0

Alerts

Name	Risk Level	Number of Instances
CSP: Wildcard Directive	Medium	2
Content Security Policy (CSP) Header Not Set	Medium	2
Cross-Domain Misconfiguration	Medium	9
Missing Anti-clickjacking Header	Medium	2
Sub Resource Integrity Attribute Missing	Medium	2
Cross-Domain JavaScript Source File Inclusion	Low	2
Permissions Policy Header Not Set	Low	5
Server Leaks Information via "X-Powered-By" HTTP Response Header Field(s)	Low	9
X-Content-Type-Options Header Missing	Low	5
Information Disclosure - Suspicious Comments	Informational	13
Modern Web Application	Informational	2
Non-Storable Content	Informational	3
Storable and Cacheable Content	Informational	5
Storable but Non-Cacheable Content	Informational	2

ภาพที่ 3-10 หน้าการแสดงผลรายงานความปลอดภัยในรูปแบบ HTML ของ OWASP ZAP

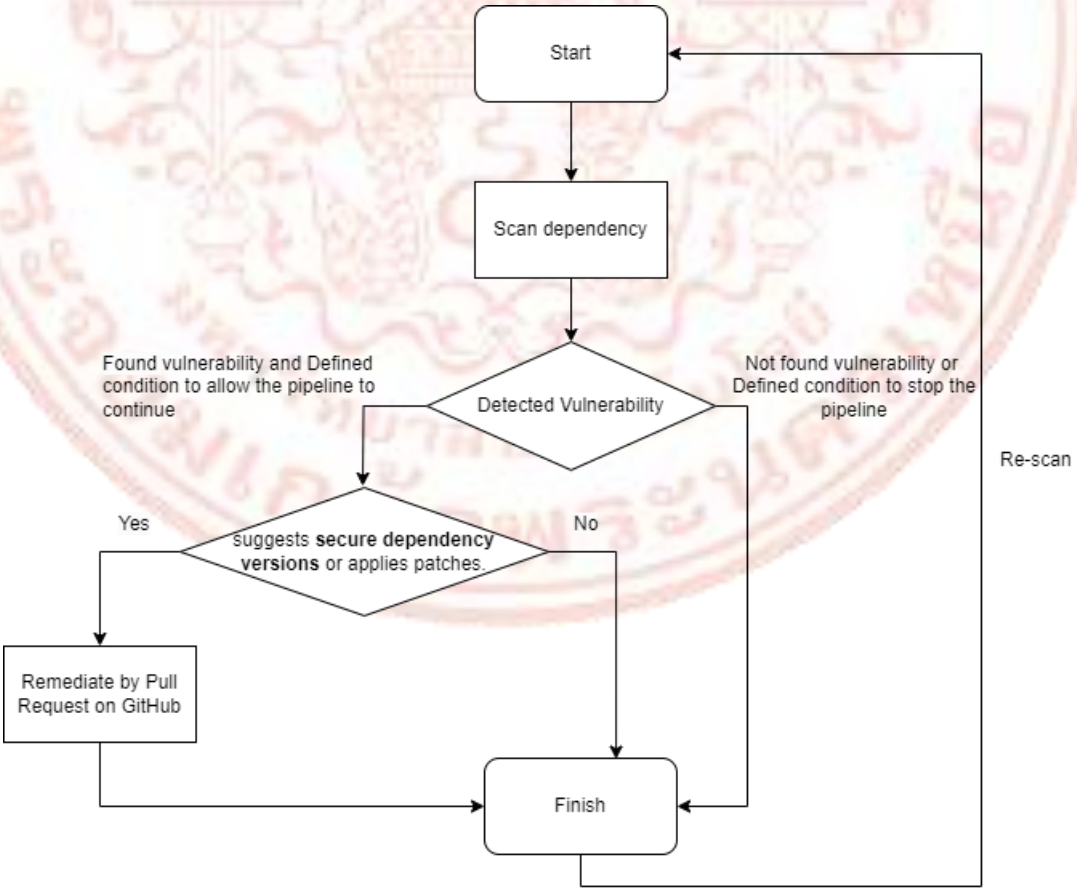
```
stage('OWASP ZAP Scan-ls') {
    steps {
        script {
            sh '''
                docker exec devsecops-zap zap-baseline.py -t http://20.212.250.197:1001 \
                -r /zap/wrk/zapreport-1-medium-code-apple-music-js-jk.html -I || true

            '''
            sh 'docker cp devsecops-zap:/zap/wrk/zapreport-1-medium-code-apple-music-js-jk.html \
            /mnt/zap-reports/zapreport-1-medium-code-apple-music-js-jk.html'
        publishHTML([
                reportName: 'zapreport-1-medium-code-apple-music-js-jenkins',
                reportDir: '/mnt/zap-reports/',
                reportFiles: 'zapreport-1-medium-code-apple-music-js-jk.html',
                keepAll: true,
                allowMissing: true,
                alwaysLinkToLastBuild: true
            ])
        }
    }
}
```

ภาพที่ 3-11 การกำหนดค่า OWASP ZAP ในเวิร์คโฟลว์ Jenkinsfile

3.3.4 การพัฒนาขั้นตอนการสร้างที่ปลอดภัย (Secure Build Stage CI/CD pipeline)

ในการทดลองนี้ จะทำการรัน CI/CD ไปป์ไลน์ความปลอดภัยแบบหลายชั้นสองครั้ง (เน้นที่ Build Stage) เพื่อลดช่องโหว่ โดยการรันครั้งแรกเป็นการระบุและแก้ไขช่องโหว่ของ Dependency ด้วย Snyk จากนั้น การรันครั้งที่สอง Trivy จะตรวจสอบจำนวนช่องโหว่อีกครั้งหลังจากการแก้ไขของ Snyk เนื่องจากเป็นการประเมินประสิทธิภาพของการลดความเสี่ยง ซึ่งมีรายละเอียดดังนี้

3.3.4.1 การรันครั้งแรก Snyk จะทำงานหลัง Build Stage เสร็จสิ้น โดยภาพที่ 3-12 ที่แสดงกระบวนการทำงาน นอกจากนี้ เมื่อ Snyk ทำการตรวจพบช่องโหว่ ผู้พัฒนาสามารถกำหนดเงื่อนไขการทำงานของ Snyk ใน Snyk Stage ของ CI/CD ไปป์ไลน์ได้ ตัวอย่างเช่น เมื่อ Snyk ตรวจพบช่องโหว่ ให้ไปป์ไลน์หยุดดำเนินการทันที แต่ในการทดลองนี้ ต้องการให้ช่องโหว่ได้รับการแก้ไข ดังนั้นจึงกำหนดให้ไปป์ไลน์ได้ทำงานต่อไปแม้พบช่องโหว่ จากนั้น Snyk จะแนะนำเวอร์ชันใหม่ของ Dependency เพื่อแทนที่เวอร์ชันเก่า โดยจะทำการสร้าง GitHub Pull Request ขึ้นมาโดยอัตโนมัติ ซึ่งต้องการให้ผู้พัฒนาคลิกยืนยันการ Pull Request นี้อีกครั้ง ทำให้ทีมพัฒนา ทีม DevOps หรือทีมอื่น ๆ ที่เกี่ยวข้องสามารถตรวจสอบหรือทบทวนการแก้ไขนี้ได้ร่วมกันอีกครั้ง เพื่อลดความผิดพลาดที่อาจเกิดขึ้น

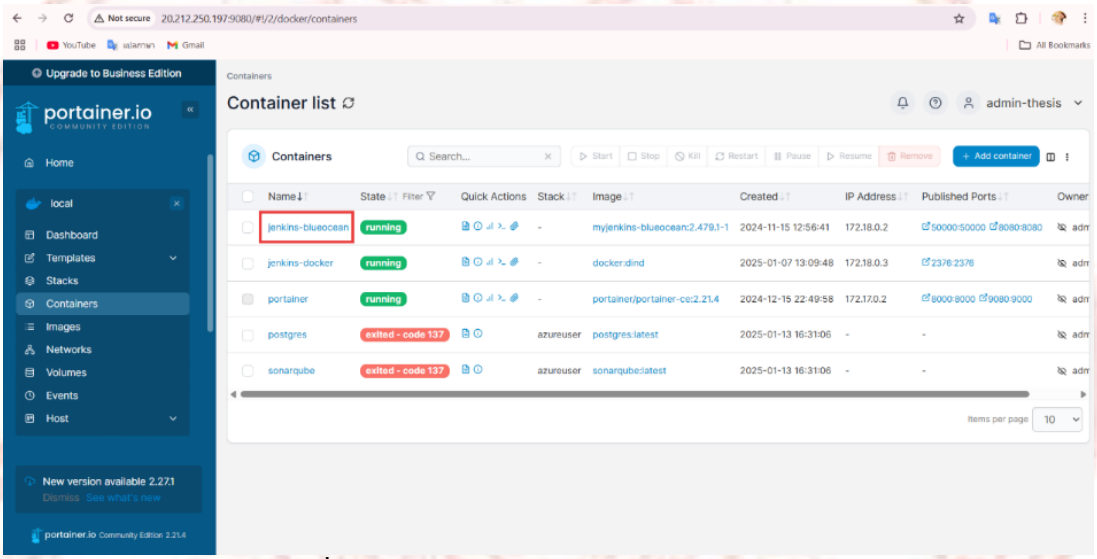


ภาพที่ 3-12 กระบวนการทำงานของ Snyk

3.3.4.2 การรันครั้งที่สอง Trivy จะตรวจจับจำนวนช่องโหว่หลังจากที่ Snyk แก้ไขเสร็จ เพื่อเป็นการตรวจสอบผลลัพธ์ความปลอดภัยใน Build Stage

3.3.3 การตรวจสอบทรัพยากรอย่างต่อเนื่อง (Resource Monitoring)

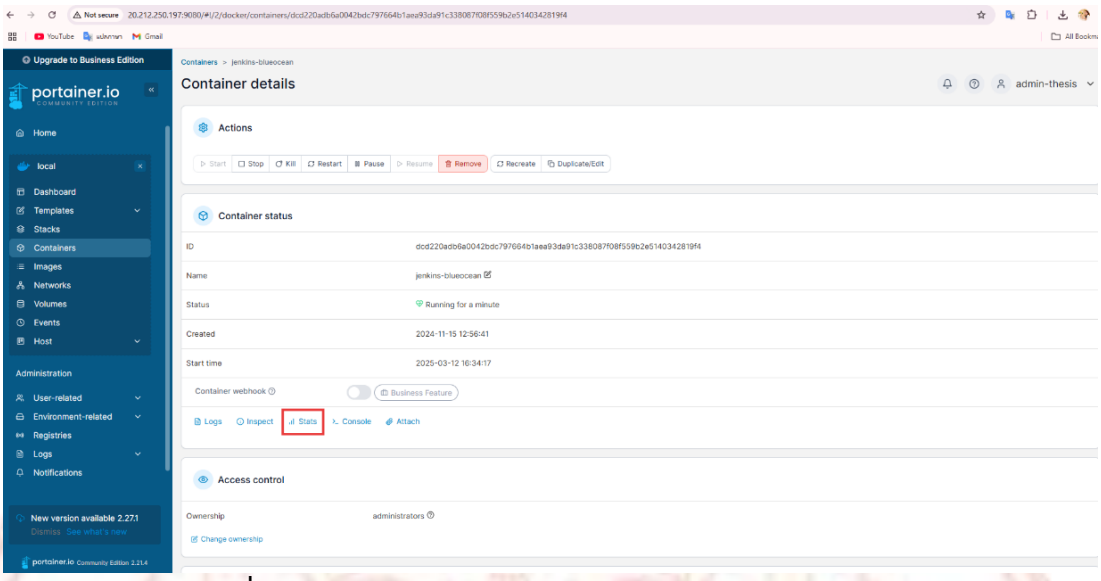
ดำเนินการโดยใช้ Portainer โดยเริ่มจากการเลือกคอนเทนเนอร์ที่ต้องการตรวจสอบ ในที่นี้ ให้เลือก Container Jenkins ดังแสดงในภาพที่ 3-13 ที่แสดงรายการ Container ทั้งหมดที่อยู่ในเครื่องคอมพิวเตอร์เสมือน จากภาพยังทำให้เห็นสถานะของแต่ละ Container อีกด้วย ซึ่งเมื่อแสดงว่า “running” จะหมายถึงว่า Container กำลังทำงานอยู่หรือพร้อมใช้งานได้ปกติ



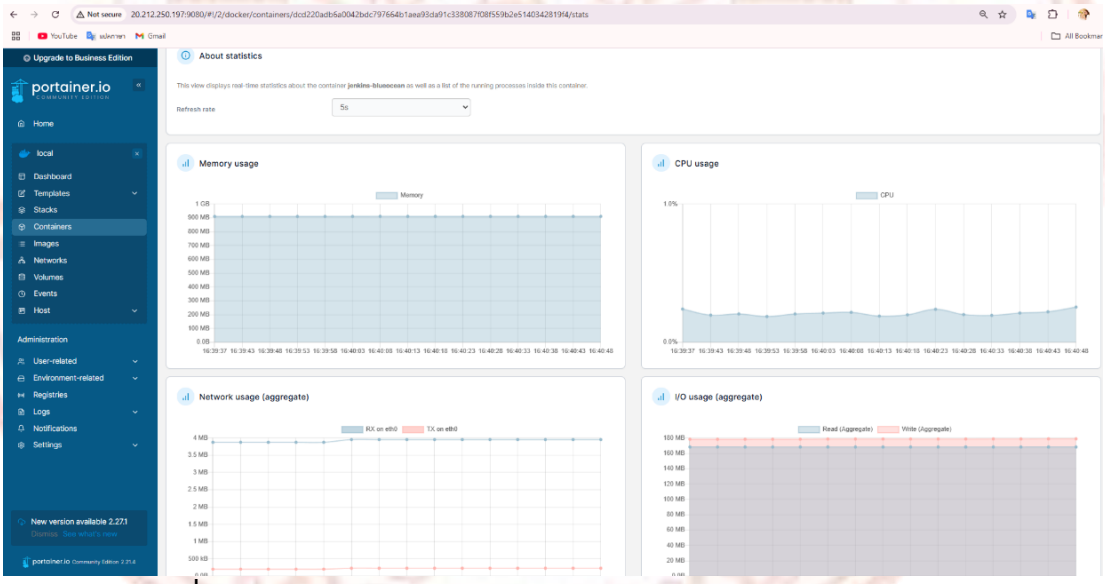
ภาพที่ 3-13 หน้ารายการ Container ของ Portainer

เมื่อเลือก Container Jenkins แล้ว ให้เปิดเมนู “Stats” สำหรับ Monitor ทรัพยากรต่าง ๆ เพื่อตรวจสอบการใช้ทรัพยากรแบบเรียลไทม์ ดังแสดงในภาพที่ 3-14 และภาพแสดงผลทรัพยากรทั้งหมดในรูปแบบของ Dashboard ดังแสดงในภาพที่ 3-15

นอกจากนี้ ภาพที่ 3-14 ยังแสดงเมนูการจัดการ Container ซึ่งอยู่ด้านบนของภาพที่แสดงตัวเลือกในการดำเนินการต่าง ๆ กับ Container เช่น หยุด (Stop) รีสตาร์ท (Restart) และอื่น ๆ ซึ่งช่วยให้ผู้ใช้สามารถควบคุม จัดการได้อย่างสะดวก



ภาพที่ 3-14 หน้ารายละเอียด Container Jenkins ของ Portainer



ภาพที่ 3-15 หน้ารายละเอียดทรัพยากร Container Jenkins ของ Portainer

การทดลองนี้ ดำเนินการตรวจสอบการใช้ทรัพยากรหน่วยความจำ (Memory) และการประมวลผล (CPU) โดยสังเกตแนวโน้มของกราฟทรัพยากรทั้งสองประเภท และทำการเทียบกับเวลาขณะที่ไปป์ไลน์ทั้งสองรูปแบบกำลังดำเนินการ รายละเอียดการทดสอบมีดังนี้

3.3.3.1 เปรียบเทียบแนวโน้มการใช้ทรัพยากรของ CI/CD ไปป์ไลน์ที่ไม่มีความปลอดภัย และ CI/CD ไปป์ไลน์ที่มีความปลอดภัยแบบหลายชั้น

ทำการสังเกตการใช้เวลาดำเนินการ หน่วยความจำ และ CPU ในระหว่างการดำเนินการของแต่ละขั้นตอนในไปป์ไลน์ทั้งสองรูปแบบ พร้อมวิเคราะห์การเปลี่ยนแปลงของทรัพยากร ที่อาจเพิ่มขึ้นหรือลดลงเมื่อมีการเพิ่มขั้นตอนรักษาความปลอดภัย

3.3.3.2 วิเคราะห์แนวโน้มกราฟการดำเนินการของไปป์ไลน์ทั้งสองรูปแบบ

ตรวจสอบขั้นตอนที่มีการใช้ทรัพยากรสูงสุด (Peak Load) ของแต่ละไปป์ไลน์ พร้อมวิเคราะห์ความสัมพันธ์ระหว่างการเพิ่มขึ้นของทรัพยากรกับการลดความเสี่ยงด้านความปลอดภัย

3.3.3.3 วิเคราะห์แนวโน้มกราฟก่อนและหลังการแก้ไขช่องโหว่ใน Build Stage

ตรวจสอบ วิเคราะห์ และเปรียบเทียบการใช้ทรัพยากรต่าง ๆ ทั้งก่อนแก้ไขและหลังแก้ไขช่องโหว่ของ Dependencies ใน Build Stage พร้อมวิเคราะห์ความคุ้มค่าของการแลกเปลี่ยนทรัพยากรที่เพิ่มขึ้นกับความปลอดภัยที่ได้รับ

บทที่ 4
ผลการวิจัย

ในบทนี้ได้นำเสนอผลการวิจัยการเปรียบเทียบประสิทธิภาพของ CI/CD ไปป์ไลน์ทั้งสองรูปแบบ โดยได้แบ่งหัวข้อหลัก ๆ ได้ทั้งหมดสามหัวข้อ ได้แก่ ผลการใช้ทรัพยากรของ CI/CD ไปป์ไลน์ความปลอดภัยแบบหลายชั้น ผลการใช้ทรัพยากรของ Build Stage และผลการตรวจจับช่องโหว่ใน Build Stage ซึ่งมีรายละเอียดดังต่อไปนี้

4.1 ผลการใช้ทรัพยากรของ CI/CD ไปป์ไลน์ความปลอดภัยแบบหลายชั้น

4.1.1 ผลการใช้ทรัพยากรเวลาดำเนินการ (Execution Time)

ในการทดลองนี้ ได้ทำการตรวจสอบเวลาดำเนินการในแต่ละขั้นตอนของไปป์ไลน์ CI/CD ทั้งสองรูปแบบ ได้แก่ ไปป์ไลน์ที่ไม่มีความปลอดภัย (Unsecured CI/CD Pipeline) และไปป์ไลน์ที่มีความปลอดภัยแบบหลายชั้น (Layered Secure CI/CD Pipeline) สำหรับโค้ด Node.js ทั้งสามขนาด ได้แก่ เล็ก กลาง และใหญ่ โดยผลการตรวจสอบเวลาดำเนินการได้ถูกนำเสนอในภาพที่ 4-1, 4-2, และ 4-3 ซึ่งแสดงถึงตัวอย่างของเวลาดำเนินการของไปป์ไลน์ทั้งสองรูปแบบสำหรับโค้ดขนาดเล็ก กลาง และใหญ่ ตามลำดับ โดยแต่ละภาพแสดงถึงผลการตรวจสอบเวลาดำเนินการของหนึ่งในสามไฟล์โค้ดของแต่ละขนาด

จากการวิเคราะห์ผลการตรวจสอบเวลาดำเนินการจากภาพดังกล่าว พบว่า การเพิ่มเครื่องมือรักษาความปลอดภัยในแต่ละขั้นตอนของไปป์ไลน์ที่มีความปลอดภัยแบบหลายชั้น ส่งผลให้เวลาดำเนินการโดยรวมเพิ่มขึ้นอย่างชัดเจน ซึ่งแต่ละขั้นตอนความปลอดภัยที่เพิ่มเข้าไปเหล่านี้ มีความจำเป็นในการรักษาความปลอดภัยของซอฟต์แวร์ แต่ก็ส่งผลให้เวลาดำเนินการโดยรวมเพิ่มขึ้น เมื่อเปรียบเทียบกับไปป์ไลน์ที่ไม่มีความปลอดภัย พบว่าโดยเฉลี่ยแล้ว ไปป์ไลน์ที่มีความปลอดภัยแบบหลายชั้น ใช้เวลาดำเนินการเพิ่มขึ้นประมาณ 2 เท่าสำหรับโค้ดทุกขนาด

อย่างไรก็ตาม ในการทดลองนี้ พบว่าไฟล์โค้ดขนาดใหญ่หนึ่งในสามไฟล์ มีเวลาดำเนินการเพิ่มขึ้นผิดปกติ ดังแสดงในภาพที่ 4-4 ซึ่งส่งผลให้ค่าเฉลี่ยรวมของเวลาดำเนินการสำหรับโค้ดขนาดใหญ่มีความคลาดเคลื่อนเล็กน้อย

Metric	Unsecured Pipeline	Layered Secure Pipeline
Time (Checkout code Stage)	1.72s	2.09s
Time (Sonarqube)	-	1.21m
Time (Build Stage)	2.11m	2.10m
Time (Snyk)	-	59s
Time (Push Stage)	41s	39s
Time (Trivy)	-	30s
Time (Deploy Stage)	1m 59s	41s
Time (ZAP)	-	1m 9s
Deployment Time (Total)	4 min 57 sec	7 min 41 sec

ภาพที่ 4-1 การเปรียบเทียบเวลาดำเนินการของไปป์ไลน์ทั้งสองรูปแบบของโค้ดขนาดเล็ก

Metric	Unsecured Pipeline	Layered Secure Pipeline
Time (Checkout code Stage)	1.99s	1.76s
Time (Sonarqube)	-	1m 17s
Time (Build Stage)	2m 7s	2m 25s
Time (Snyk)	-	1m 1s
Time (Push Stage)	33s	34s
Time (Trivy)	-	26s
Time (Deploy Stage)	40s	41s
Time (ZAP)	-	47s
Deployment Time (Total)	3 min 29 sec	7 min 19 sec

ภาพที่ 4-2 การเปรียบเทียบเวลาดำเนินการของไปป์ไลน์ทั้งสองรูปแบบของโค้ดขนาดกลาง

Metric	Unsecured Pipeline	Layered Secure Pipeline
Time (Checkout code Stage)	3s	2s
Time (Sonarqube)	-	1m 15s
Time (Build Stage)	1m 48s	1m 47s
Time (Snyk)	-	1m 5s
Time (Push Stage)	1m 1s	57s
Time (Trivy)	-	56s
Time (Deploy Stage)	54s	46s
Time (ZAP)	-	47s
Deployment Time (Total)	3.55min	7.42min

ภาพที่ 4-3 การเปรียบเทียบเวลาดำเนินการของไปป์ไลน์ทั้งสองรูปแบบของโค้ดขนาดใหญ่

Metric	Unsecured Pipeline	Layered Secure Pipeline
Time (Checkout code Stage)	2.34s	6.9s
Time (Sonarqube)	-	2m 28s
Time (Build Stage)	10m	12m
Time (Snyk)	-	2m 4s
Time (Push Stage)	36s	36s
Time (Trivy)	-	35s
Time (Deploy Stage)	40s	50s
Time (ZAP)	-	1m 9s
Deployment Time (Total)	12min	20.04min

ภาพที่ 4-4 การเปรียบเทียบเวลาดำเนินการของไปป์ไลน์ทั้งสองรูปแบบของไฟล์โค้ดขนาดใหญ่ที่ผิดปกติ

ผลลัพธ์ที่นำเสนอในตารางที่ 4-1 คือเวลาเฉลี่ยรวมจากการดำเนินการกับโค้ดทุกขนาด โดยการใช้ทรัพยากรเวลาดำเนินการของไปป์ไลน์ความปลอดภัยแบบหลายชั้น ซึ่งเวลาที่เพิ่มขึ้นนี้เป็นผลมาจากการบูรณาการขั้นตอนการสแกนโค้ดด้วย SonarQube การตรวจสอบช่องโหว่ด้วย Snyk (เฉพาะกรณีที่ตั้งค่าให้ Snyk ดำเนินการต่อเมื่อพบช่องโหว่) การตรวจสอบช่องโหว่ด้วย Trivy และการ

ทดสอบเจาะระบบความปลอดภัยด้วย OWASP ZAP ซึ่งแต่ละขั้นตอนนั้น ได้ใช้เวลาเพิ่มเติมอย่างมีนัยสำคัญ

ตารางที่ 4-1 การเปรียบเทียบทรัพยากรเวลาที่ไปป์ไลน์ทั้งสองรูปแบบใช้

ขนาดโค้ด	Unsecured Pipeline	Layered secure Pipeline	ค่าความแตกต่าง
เล็ก	176.657×10^9	378.830×10^9	เพิ่มขึ้น 2.14 เท่า
กลาง	202.997×10^9	394.653×10^9	เพิ่มขึ้น 1.94 เท่า
ใหญ่	356.607×10^9	713.670×10^9	เพิ่มขึ้น 2 เท่า

จากการวิเคราะห์พบว่าแต่ละขนาดของโค้ดมีผลกระทบต่อเวลาดำเนินการในระดับที่ใกล้เคียงกัน โดยโค้ดขนาดเล็กเพิ่มขึ้นร้อยละ 114.44 โค้ดขนาดกลางเพิ่มขึ้นร้อยละ 94.41 และโค้ดขนาดใหญ่เพิ่มขึ้นร้อยละ 100.13 แสดงให้เห็นว่า ไม่ว่าโค้ดขนาดใหญ่ที่มี Dependencies จำนวนมาก หรือโค้ดขนาดเล็กที่อาจมี Dependencies ที่ซับซ้อนกว่า ทำให้ใช้เวลาดำเนินการเพิ่มขึ้น อย่างไรก็ตามแม้ว่าเวลาดำเนินการจะเพิ่มขึ้น แต่การเพิ่มขึ้นนี้ถือว่าอยู่ในระดับที่ยอมรับได้ เมื่อเทียบกับประโยชน์ด้านความปลอดภัยที่ได้รับในระยะยาว โดยไปป์ไลน์ที่มีความปลอดภัยแบบหลายชั้นสามารถลดความเสี่ยงจากช่องโหว่ได้อย่างมีประสิทธิภาพในหลาย ๆ ขั้นตอนของกระบวนการ CI/CD ซึ่งเป็นสิ่งที่ไปป์ไลน์แบบธรรมดาไม่สามารถทำได้

4.1.2 ผลการใช้ทรัพยากรหน่วยความจำ (Memory)

จากการสังเกตจากการทดลอง โดยปกติแล้วไปป์ไลน์จะใช้หน่วยความจำสูงขึ้นในช่วง Build stage หาก Image มีขนาดใหญ่ แต่เมื่อมีการบูรณาการเครื่องมือรักษาความปลอดภัยเข้ากับ CI/CD ไปป์ไลน์นั้น ส่งผลกระทบกับการใช้ทรัพยากรหน่วยความจำอย่างชัดเจน โดยจุดเริ่มการใช้หน่วยความจำในทุกขนาดของโค้ดมีแนวโน้มที่ต่ำ และหลังจากนั้น จะมีแนวโน้มที่เพิ่มสูงขึ้นในช่วงขั้นตอนสุดท้ายของกระบวนการไปป์ไลน์ ได้แก่ ขั้นตอน Trivy, Deploy, OWASP ZAP เนื่องจากการดำเนินการของทั้งสามขั้นตอนนี้ มีความต้องการในการโหลดข้อมูลจำนวนมากเข้าสู่หน่วยความจำ ส่งผลให้ปริมาณการใช้ทรัพยากรหน่วยความจำเพิ่มขึ้นประมาณร้อยละ 22 เมื่อเทียบกับไปป์ไลน์แบบธรรมดาที่ไม่มีความปลอดภัย โดยการใช้งานหน่วยความจำเพิ่มขึ้นจากช่วง 1.4 - 2.6 กิกะไบต์ (Gigabyte) โดยเฉพาะขั้นตอน OWASP ZAP ที่เป็นขั้นตอนที่มีการใช้หน่วยความจำสูงที่สุด

แต่ในขณะเดียวกัน ไปป์ไลน์แบบธรรมดาที่ไม่มีมาตรการรักษาความปลอดภัย แสดงให้เห็นถึงกราฟการใช้ทรัพยากรหน่วยความจำที่มีแนวโน้มค่อนข้างคงที่ ดังแสดงภาพที่ 4-5 ซึ่งแสดงให้เห็นถึงความแตกต่างอย่างชัดเจนระหว่างสองรูปแบบของไปป์ไลน์

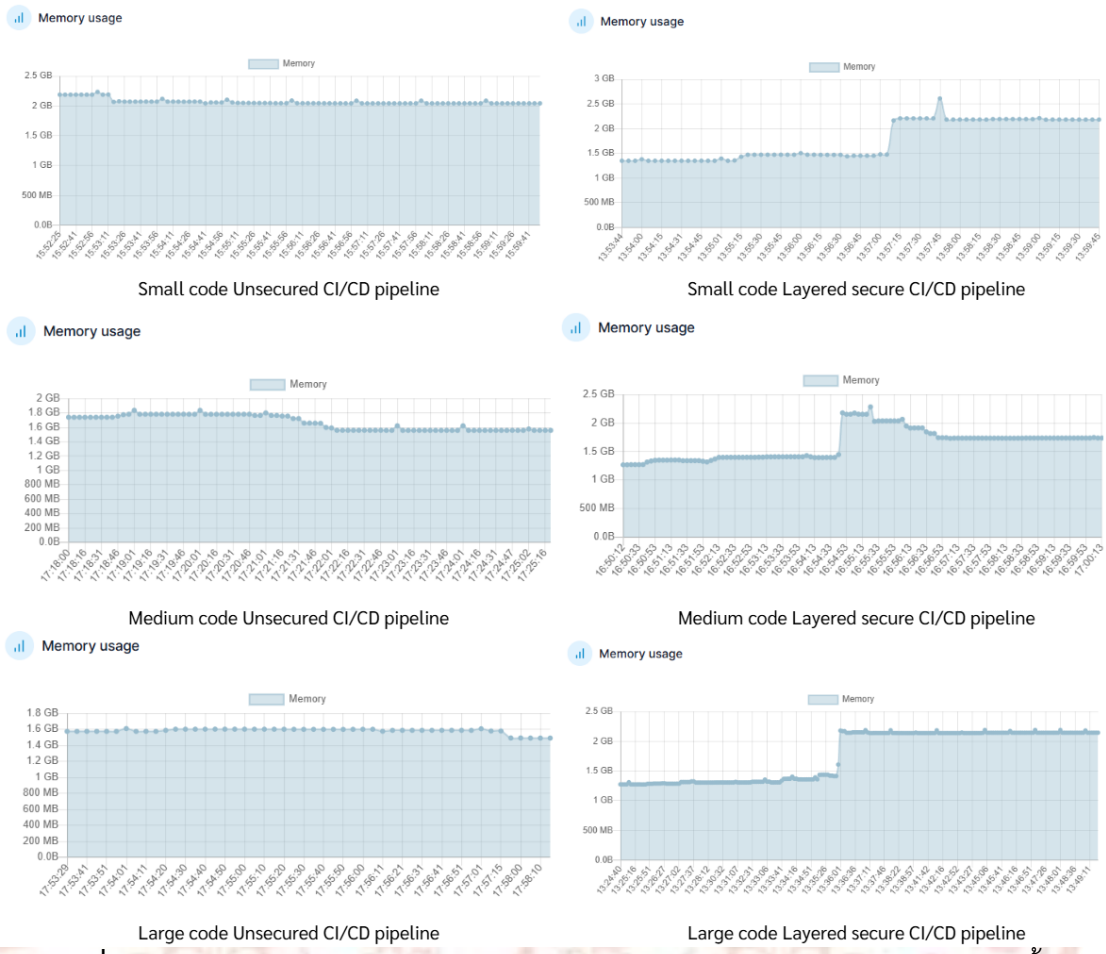
ผู้วิจัยได้ทำการสุ่มรูปภาพกราฟการใช้หน่วยความจำของโค้ดแต่ละขนาดมาแสดง โดยสุ่มขนาดโค้ดละ 1 รูป เพื่อเปรียบเทียบการใช้หน่วยความจำระหว่างไปป์ไลน์ความปลอดภัยแบบหลายชั้นและไปป์ไลน์แบบธรรมดา จากการสังเกตพบว่า ทุกขนาดโค้ดแสดงให้เห็นถึงในช่วงเริ่มต้นของไปป์ไลน์ความปลอดภัยแบบหลายชั้นมีการใช้หน่วยความจำที่ต่ำกว่าไปป์ไลน์แบบธรรมดา ก่อนที่จะเพิ่มขึ้นอย่างชัดเจนในช่วงขั้นตอนสุดท้ายของกระบวนการ ซึ่งหมายความว่าผู้พัฒนาสามารถใช้ประโยชน์จากหน่วยความจำที่มีอยู่ได้อย่างมีประสิทธิภาพในช่วงเริ่มต้นของไปป์ไลน์

นอกจากนี้ จากการสังเกตจากการทดลอง การตรวจสอบการใช้ทรัพยากรหน่วยความจำของ Container Jenkins ผ่าน Portainer พบว่า ค่าการใช้ทรัพยากรในช่วงเริ่มต้นของการดำเนินการไปป์ไลน์ CI/CD ไม่สามารถกำหนดให้เป็นค่าคงที่ได้อย่างแม่นยำ หรืออาจมีความคลาดเคลื่อนเกิดขึ้นได้ ซึ่งมีสาเหตุมาจากปัจจัย ดังนี้

- การใช้ทรัพยากรของเครื่องคอมพิวเตอร์เสมือนที่ใช้ในการทดลองนี้ ขึ้นอยู่กับการใช้งานทรัพยากรในขณะนั้นของระบบปฏิบัติการและ Container อื่น ๆ ที่กำลังทำงานบนเครื่องคอมพิวเตอร์เสมือนอยู่ ซึ่งส่งผลให้ค่าการใช้ทรัพยากรหน่วยความจำของคอนเทนเนอร์ Jenkins มีความผันผวน
- ทรัพยากรของเครื่องแม่ข่าย (Host Server) จากไมโครซอฟต์ อาซัวร์ที่ให้บริการเครื่องคอมพิวเตอร์เสมือนแก่ผู้ใช้ รวมถึงการทดลองนี้ จะถูกแจกจ่ายให้กับลูกค้าหลายราย ซึ่งส่งผลให้ทรัพยากรที่ได้รับในแต่ละช่วงเวลาอาจแตกต่างกันไป ทำให้ค่าการใช้ทรัพยากรหน่วยความจำของคอนเทนเนอร์ Jenkins ในช่วงเริ่มต้นของการดำเนินการไปป์ไลน์ CI/CD ไม่สามารถกำหนดให้เป็นค่าคงที่ได้ในแต่ละครั้ง

อย่างไรก็ตาม ในการทดลองนี้ ได้ทำการตรวจสอบการใช้ทรัพยากรของคอนเทนเนอร์ Jenkins โดยเฉพาะ ซึ่งช่วยลดผลกระทบจากปัจจัยภายนอกที่อาจส่งผลต่อการใช้ทรัพยากรโดยรวมของเครื่องคอมพิวเตอร์เสมือน ทำให้ความคลาดเคลื่อนที่เกิดขึ้นยังอยู่ในระดับที่ยอมรับได้ และไม่ส่งผลกระทบต่อผลการวิเคราะห์โดยรวมของการทดลอง

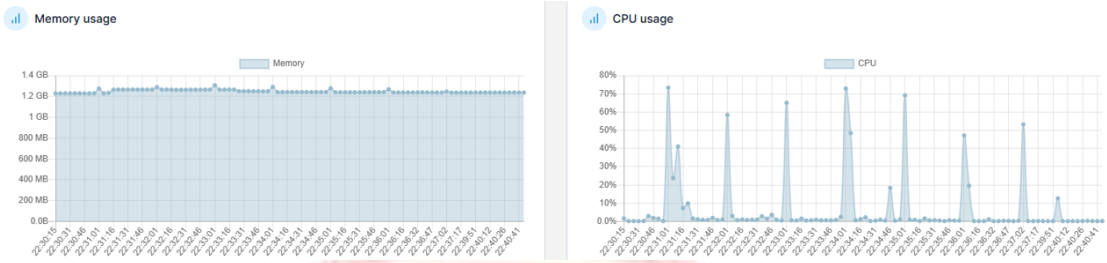
โดยสรุปแล้ว แม้ว่าการใช้หน่วยความจำจะเพิ่มขึ้น แต่เป็นการแลกเปลี่ยนเพื่อความปลอดภัยและการลดช่องโหว่ในกระบวนการ CI/CD ไปป์ไลน์ ที่มีขั้นตอนตั้งแต่การพัฒนาซอฟต์แวร์ จนไปถึงการส่งมอบให้กับลูกค้า ดังนั้นองค์กรสามารถบริหารทรัพยากรหน่วยความจำนี้ได้อย่างมีประสิทธิภาพ โดยใช้การจัดสรรที่เหมาะสมและปรับแต่งไปป์ไลน์ให้เหมาะสมกับขนาดของโค้ด



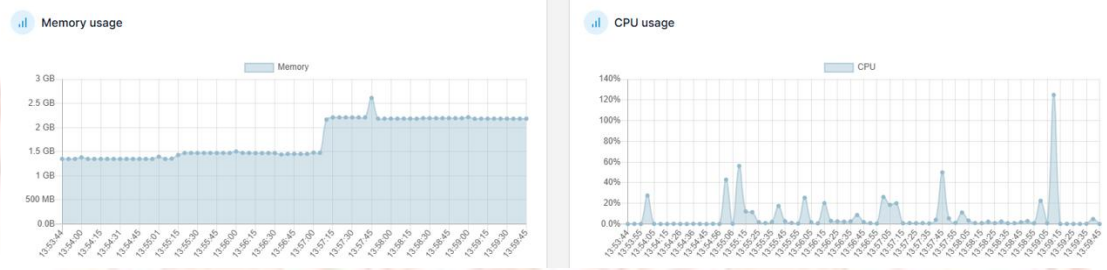
ภาพที่ 4-5 แนวโน้มการใช้ทรัพยากรหน่วยความจำของไปป์ไลน์ความปลอดภัยแบบหลายชั้น

4.1.3 ผลการใช้ทรัพยากรการประมวลผล (CPU)

ผลจากการทดลองแสดงให้เห็นว่า การใช้ทรัพยากรหน่วยความจำไม่ได้ส่งกระทบโดยตรงต่อการใช้ CPU เสมอไป ซึ่งขัดแย้งกับความสัมพันธ์ที่คาดการณ์ไว้ในตอนแรก โดยทั่วไปแล้ว การเพิ่มขึ้นของการใช้หน่วยความจำ จะทำให้การใช้ทรัพยากร CPU เพิ่มขึ้นไปด้วย เนื่องจากการดำเนินการอาจต้องใช้ CPU ในการจัดการข้อมูลที่อยู่ในหน่วยความจำ อย่างไรก็ตาม ในการทดลองนี้ พบว่าบางขั้นตอนของไปป์ไลน์ที่มีการใช้หน่วยความจำสูง ก็ไม่ได้มีการใช้ CPU สูงตามไปด้วย ดังแสดงในภาพที่ 4-6 และภาพที่ 4-7 ซึ่งเป็นกราฟที่แสดงการใช้หน่วยความจำและ CPU ในช่วงเวลาเดียวกันของไปป์ไลน์ทั้งสองรูปแบบ



ภาพที่ 4-6 กราฟที่แสดงการใช้หน่วยความจำและ CPU ในช่วงเวลาเดียวกันของไปป์ไลน์ที่ไม่มีความปลอดภัย



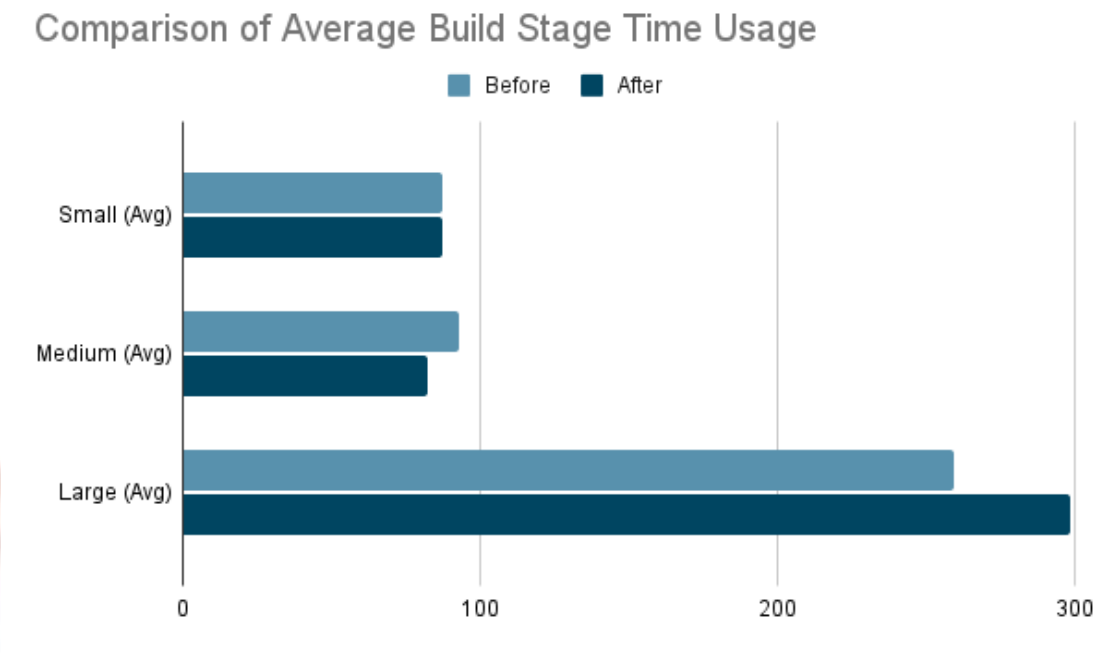
ภาพที่ 4-7 กราฟที่แสดงการใช้หน่วยความจำและ CPU ในช่วงเวลาเดียวกันของไปป์ไลน์ที่มีความปลอดภัยแบบหลายชั้น

4.2 ผลการใช้ทรัพยากรของ Build Stage

ในส่วนนี้ ได้แสดงผลการวิเคราะห์ทรัพยากรต่าง ๆ ได้แก่ เวลาดำเนินการ หน่วยความจำ และ CPU ทั้งก่อนและหลังแก้ไขช่องโหว่ใน Build Stage ด้วย Snyk มีรายละเอียดดังนี้

4.2.1 ผลการใช้ทรัพยากรเวลาดำเนินการ (Execution Time)

จากการวิเคราะห์เวลาดำเนินการใน Build Stage ของไปป์ไลน์ CI/CD ทั้งสองรูปแบบ ได้แสดงให้เห็นว่า การแก้ไขช่องโหว่ด้วย Snyk ไม่ได้ส่งผลกระทบต่อเวลาดำเนินการอย่างเห็นได้ชัด เนื่องจากการแก้ไขช่องโหว่เป็นการดำเนินการอัปเดตและปรับปรุงเวอร์ชันของ Dependencies โดยที่จำนวน Dependencies ที่ถูกอัปเดตยังคงเท่าเดิม ส่งผลให้เวลาดำเนินการก่อนหรือหลังแก้ไขช่องโหว่ ยังมีแนวโน้มที่ค่อนข้างใกล้เคียงกัน ซึ่งเห็นได้ชัดในโค้ดขนาดเล็กและขนาดกลาง ดังแสดงในภาพที่ 4-8 แต่ในขณะที่โค้ดขนาดใหญ่ในการทดลองนี้ มีไฟล์โค้ดจำนวนหนึ่งไฟล์ ที่มีการใช้ทรัพยากรเพิ่มขึ้นผิดปกติ ซึ่งส่งผลให้ค่าเฉลี่ยรวมของทรัพยากรสำหรับโค้ดขนาดใหญ่มีความคลาดเคลื่อน โดยได้บันทึกผลการใช้เวลาดำเนินการในภาพที่ 4-1, 4-2 และ 4-3 ตามขนาดของโค้ด ซึ่งได้ตรวจสอบใน Build Stage เป็นหลัก โดย Build Stage ในฝั่ง Layered secure Pipeline จะเป็นผลเวลาดำเนินการหลังจากที่แก้ไขช่องโหว่ด้วย Snyk เรียบร้อยแล้ว โดยหน่วยเวลาที่ใช้ในการวัดและบันทึกผลการดำเนินการคือนาที



ภาพที่ 4-8 การเปรียบเทียบการใช้เวลาดำเนินการก่อน-หลังการแก้ไขช่องโหว่ของ Build Stage

4.2.2 ผลการใช้งานทรัพยากรหน่วยความจำ (Memory)

จากการทดลองตรวจสอบการใช้ทรัพยากรหน่วยความจำของ Build Stage ทั้งก่อนและหลังการแก้ไขช่องโหว่ ดังแสดงในภาพที่ 4-9, 4-10 และ 4-11 ซึ่งแสดงผลลัพธ์แต่ละขนาดโค้ดทั้งสามขนาด ได้แก่ ขนาดเล็ก กลาง และใหญ่ ตามลำดับ

Metric Memory usage	Unsecured Pipeline	Layered Secure Pipeline
Checkout code Stage	[1.2 - 1.3 GB]	900 MB
Sonarqube Stage		900 MB
Build Stage	[1.2 - 1.3 GB]	[900 MB - 1 GB]
Snyk Stage		1 GB
Push Stage	[1.2 - 1.3 GB]	1 GB
Trivy		[900 MB - 1 GB]
Deploy Stage	[1.2 - 1.3 GB]	[900 MB - 1 GB}
ZAP		1.6 GB

ภาพที่ 4-9 การเปรียบเทียบการใช้หน่วยความจำของไปป์ไลน์ทั้งสองรูปแบบของโค้ดขนาดเล็ก

Metric	Unsecured Pipeline	Layered Secure Pipeline
Checkout code Stage	1.3GB	1.4GB
Sonarqube Stage		1.5GB
Build Stage	1.4GB	1.5GB
Snyk Stage		1.5GB
Push Stage	1.4GB	1.5GB
Trivy		2GB
Deploy Stage	1.4GB	2.3GB
ZAP		2.8GB

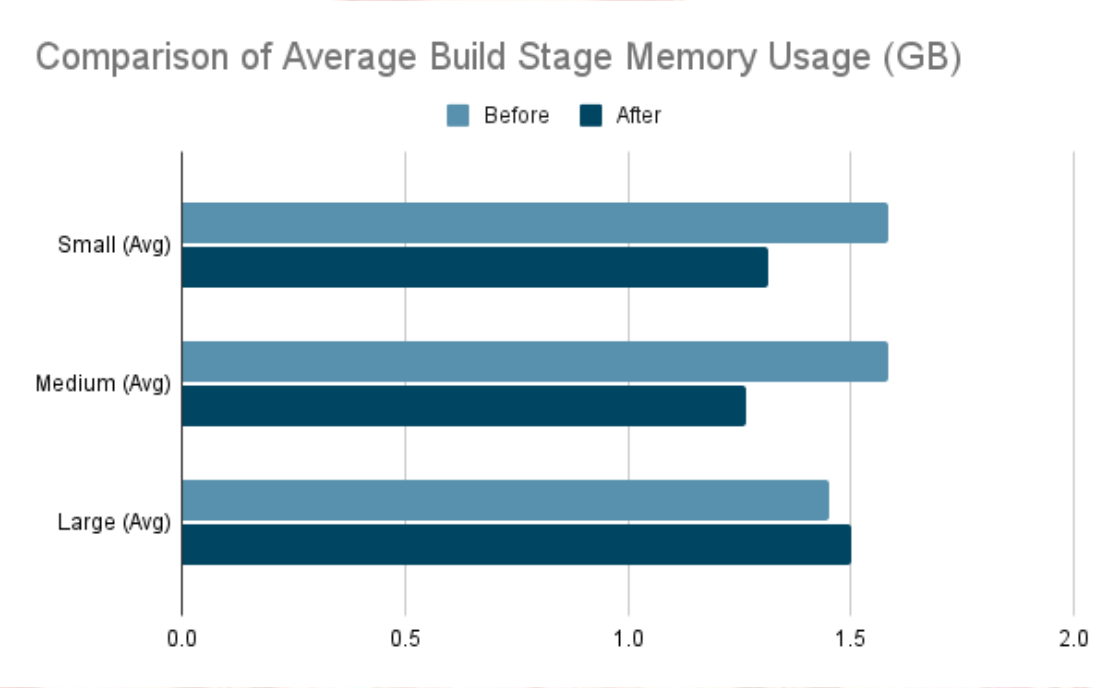
ภาพที่ 4-10 การเปรียบเทียบการใช้หน่วยความจำของไปป์ไลน์ทั้งสองรูปแบบของโค้ดขนาดกลาง

Metric	Unsecured Pipeline	Layered Secure Pipeline
Checkout code Stage	[1.7-1.8 GB]	[1.3-1.4 GB}
Sonarqube Stage		[1.3-1.4 GB}
Build Stage	[1.7-1.8 GB]	1.4 GB
Snyk Stage		1.4 GB
Push Stage	[1.7-1.8 GB]	1.4 GB
Trivy		2.2 GB
Deploy Stage	[1.7-1.8 GB]	2.2 GB
ZAP		2.3 GB

ภาพที่ 4-11 การเปรียบเทียบการใช้หน่วยความจำของไปป์ไลน์ทั้งสองรูปแบบของโค้ดขนาดใหญ่

จากภาพที่ 4-6 พบว่าหลังการแก้ไขช่องโหว่ หน่วยความจำมีการใช้น้อยลง โดยโค้ดขนาดกลางจะมีไฟล์หนึ่งที่เพิ่มขึ้นมาเพียงแค่ 100 เมกะไบต์ (MB) เท่านั้น ในขณะที่โค้ดไฟล์อื่นลดลงอย่างมีนัยสำคัญ อย่างไรก็ตาม จากการวิเคราะห์พบว่า หลังจากแก้ไขช่องโหว่แล้วการใช้ทรัพยากรหน่วยความจำโดยรวมมีแนวโน้มที่ลดลงในโค้ดทุกขนาด ดังที่แสดงในภาพที่ 4-12 ที่คำนวณผลลัพธ์ออกมาเป็นค่าเฉลี่ย โดยโค้ดขนาดใหญ่มีแนวโน้มที่เพิ่มขึ้นเพียงเล็กน้อย เนื่องจากเหตุผลที่ได้กล่าวไว้ก่อนหน้าว่าในการทดลองนี้มีไฟล์โค้ดขนาดใหญ่จำนวนหนึ่งไฟล์ที่อาจทำให้ทรัพยากรคลาดเคลื่อน นอกจากนี้ ผลลัพธ์ยังแสดงให้เห็นอีกด้วยว่า เมื่ออัปเดตเวอร์ชันของ Dependencies ทำให้ Dependencies ได้ถูกปรับปรุงให้ดีขึ้น ทำให้ใช้พื้นที่หน่วยความจำในการโหลดน้อยลงเมื่อเทียบกับเวอร์ชันก่อนหน้า ทำให้หลังจากแก้ไขช่องโหว่ โค้ดขนาดเล็กใช้ทรัพยากรหน่วยความจำลดลง

โดยเฉพาะอย่างยิ่งโค้ดขนาดกลางลดลงจากเดิมถึงร้อยละ 20 ซึ่งบ่งบอกได้ว่า Dependencies ส่วนใหญ่ได้รับการปรับปรุงอย่างมีประสิทธิภาพ ในทางกลับกัน โค้ดขนาดใหญ่กลับมีการใช้หน่วยความจำเพิ่มขึ้นร้อยละ 3.45 ซึ่งอาจเป็นผลมาจาก Dependencies ของโค้ดขนาดใหญ่มีความซับซ้อน ทำให้การอัปเดตเป็นไปได้ยาก ส่งผลให้แก้ไขได้ไม่เต็มประสิทธิภาพ



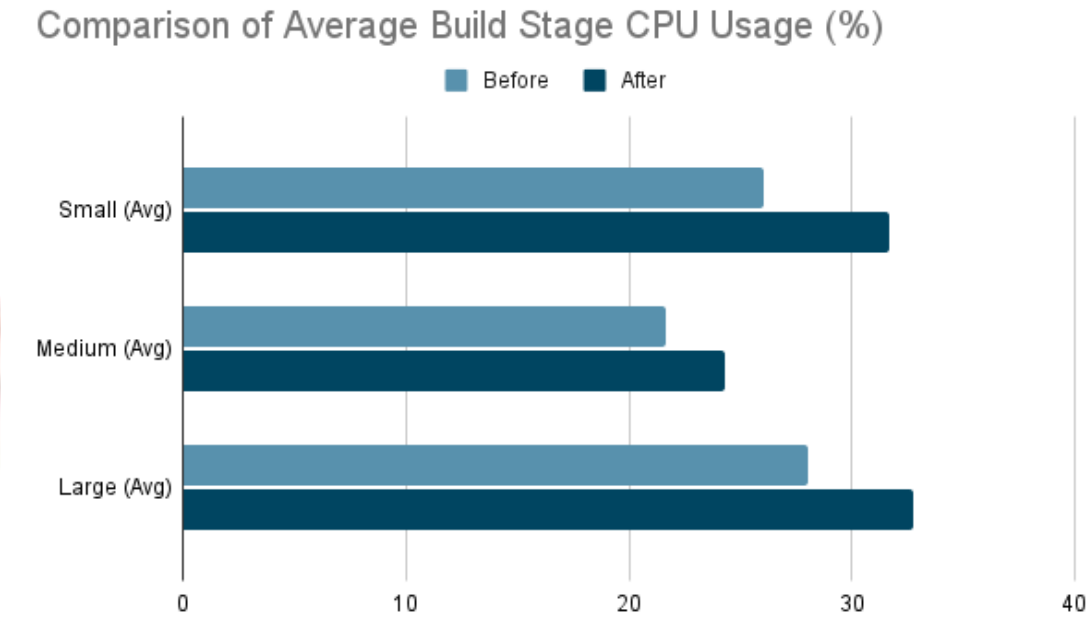
ภาพที่ 4-12 การเปรียบเทียบการใช้หน่วยความจำก่อน-หลังการแก้ไขช่องโหว่ของ Build Stage

4.2.3 ผลการใช้ทรัพยากรการประมวลผล (CPU)

ผลการทดลองแสดงให้เห็นว่า การอัปเดตเวอร์ชัน Dependencies ส่งผลให้การใช้ทรัพยากร CPU เพิ่มขึ้นในทุกขนาดของโค้ด โดยโค้ดขนาดเล็กเพิ่มขึ้นร้อยละ 21.44 โค้ดขนาดกลางเพิ่มขึ้นร้อยละ 20.25 และโค้ดขนาดใหญ่อยู่ที่ร้อยละ 17.04 ดังที่แสดงในภาพที่ 4-13 ซึ่งการเพิ่มขึ้นนี้ แสดงให้เห็นว่าการอัปเดตเวอร์ชัน Dependencies เป็นเวอร์ชันใหม่มีความซับซ้อนในการประมวลผลมากขึ้น ส่งผลให้ต้องใช้ทรัพยากร CPU เพิ่มขึ้น โดยเฉพาะอย่างยิ่งคือโค้ดขนาดใหญ่ที่มีจำนวน Dependencies มากกว่าโค้ดขนาดอื่น

อย่างไรก็ตาม แม้ว่าการใช้ CPU จะเพิ่มขึ้น แต่การเพิ่มขึ้นนี้อยู่ในระดับที่ยอมรับได้เมื่อพิจารณาถึงความปลอดภัยที่เพิ่มขึ้นจากการอัปเดต Dependencies เป็นเวอร์ชันใหม่ ซึ่งช่วยลดความเสี่ยงจากช่องโหว่ที่อาจเกิดขึ้นใน Dependencies เวอร์ชันเก่า

นอกจากนี้ ผลการทดลองยังแสดงให้เห็นว่าโค้ดขนาดกลางมีการเปลี่ยนแปลงการใช้ทรัพยากร CPU น้อยที่สุดหลังจากการแก้ไขช่องโหว่ใน Build Stage ซึ่งแสดงให้เห็นถึงประสิทธิภาพที่ดีที่สุดในการจัดการทรัพยากรระบบ โดยไม่มีผลกระทบเชิงลบต่อการใช้ CPU



ภาพที่ 4-13 การเปรียบเทียบการใช้ CPU ก่อน-หลังการแก้ไขช่องโหว่ของ Build Stage

4.3 ผลการตรวจจับช่องโหว่ใน Build Stage

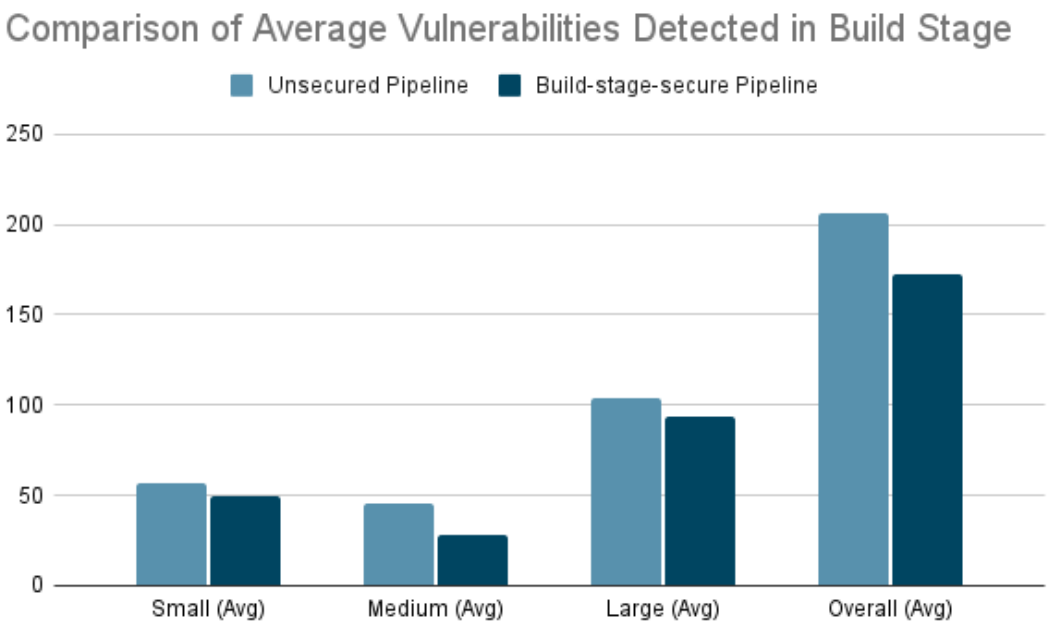
ผลการทดลองแสดงให้เห็นว่า การใช้เครื่องมือ Snyk และตรวจสอบซ้ำโดย Trivy แสดงให้เห็นถึงการลดลงของจำนวนช่องโหว่ในโค้ดทั้งหมดที่ใช้ในการทดลอง ซึ่งประกอบด้วยไฟล์จำนวน 9 ไฟล์ แบ่งเป็น 3 ขนาด ดังแสดงในภาพที่ 4-14 ซึ่งเป็นผลลัพธ์ที่ได้มาจาก Trivy โดยตรง เมื่อนำมาวิเคราะห์และคำนวณเป็นเปอร์เซ็นต์ พบว่าโค้ดขนาดเล็กมีช่องโหว่ลดลงร้อยละ 12.22 โค้ดขนาดกลางลดลดอย่างมากถึงร้อยละ 62.96 ซึ่งบ่งชี้ถึงประสิทธิภาพสูงสุดในการจัดการช่องโหว่ของ Dependencies และโค้ดขนาดใหญ่ลดลงร้อยละ 9.88 ซึ่งมีอัตราการลดลงที่น้อยที่สุดเมื่อเทียบกับโค้ดขนาดอื่น ๆ ดังแสดงภาพที่ 4-15

การลดลงของช่องโหว่เหล่านี้ ส่งผลกระทบโดยตรงต่อความปลอดภัยของซอฟต์แวร์ ซึ่งช่วยลดความเสี่ยงในการนำช่องโหว่เข้าสู่ขั้นตอนอื่นๆ ของไปป์ไลน์ และลดความเสี่ยงด้านความปลอดภัยก่อนการส่งมอบซอฟต์แวร์ให้กับลูกค้า โดยเฉพาะอย่างยิ่งโค้ดขนาดกลางที่แสดงประสิทธิภาพสูงสุดในการทดลองนี้ เนื่องจาก Dependencies หลายรายการได้รับการอัปเดตอย่างมีประสิทธิภาพ ในขณะเดียวกัน โค้ดขนาดเล็กและใหญ่ในการทดลองนี้ อาจมี Dependencies ที่ซับซ้อนและมีจำนวน

มากกว่า ได้พบปัญหาและมีข้อจำกัดในการอัปเดต ทำให้สองขนาดโค้ดนี้ มีประสิทธิภาพในการลดช่องโหว่ได้น้อยกว่าโค้ดขนาดกลาง นอกจากนี้ การลดลงของช่องโหว่ ขึ้นอยู่กับ Dependencies ที่โค้ดใช้และโครงสร้างการเขียนโปรแกรมของแต่ละผู้พัฒนาอีกด้วย โดยสรุปแล้ว การใช้เครื่องมือ Snyk ร่วมกับ Trivy ในขั้นตอน Build Stage เป็นวิธีการที่มีประสิทธิภาพในการลดช่องโหว่และเพิ่มความปลอดภัยให้กับซอฟต์แวร์ โดยเฉพาะอย่างยิ่งในโค้ดที่มี Dependencies ที่ได้รับการจัดการอย่างมีประสิทธิภาพ

Size of code	Number of Vulnerabilities Detected Before remediating	Number of Vulnerabilities Detected After remediating	Difference	Percentage
	46	**44**	2	4.347826087
	High 40 , Critical 6	High 38 , Critical 6		
Small	**79**	**76**	3	3.797468354
	High 63 , Critical 16	High 60 , Critical 16		
	47	**31**	16	34.04255319
	High 31 , Critical 16	High 24 , Critical 7		
	87	**2**	85	97.70114943
	High 40 , Critical 6	High 38 , Critical 6		
Medium	**21**	**19**	2	9.523809524
	High 19 , Critical 2	High 17 , Critical 2		
	27	**22**	5	18.51851852
	High 23 , Critical 4	High 18 , Critical 4		
	7	**3**	4	57.14285714
Large	High 7 , Critical 0	High 3 , Critical 0		
	27	**11**	16	59.25925926
	High 17 , Critical 10	High 5 , Critical 6		
	279	**268**	11	3.94265233
	High 216 , Critical 63	High 207 , Critical 61		

ภาพที่ 4-14 จำนวนช่องโหว่ก่อนและหลังแก้ไขของ Build Stage



ภาพที่ 4-15 การเปรียบเทียบช่องโหว่ใน Build Stage

บทที่ 5
สรุปผลการวิจัยและข้อเสนอแนะ

ในบทนี้ได้สรุปผลการวิจัยและข้อเสนอแนะของการศึกษาเปรียบเทียบ CI/CD ไปป์ไลน์ทั้งสองรูปแบบพร้อมข้อจำกัดของงานวิจัยครั้งนี้ โดยมีรายละเอียดดังนี้

5.1 สรุปผลการวิจัย

งานวิจัยนี้เปรียบเทียบไปป์ไลน์ CI/CD สองรูปแบบ ได้แก่ CI/CD ไปป์ไลน์ที่ไม่มีความปลอดภัยและที่มีความปลอดภัยแบบหลายชั้น โดยเน้นการลดช่องโหว่ในขั้นตอน Build Stage และตรวจสอบเวลาดำเนินการ รวมถึงการใช้ทรัพยากรหน่วยความจำและ CPU ด้วย Portainer สรุปผลได้ดังนี้

5.1.1 การตรวจสอบการใช้ทรัพยากรของไปป์ไลน์ความปลอดภัยแบบหลายชั้น

โดยงานวิจัยนี้ได้บูรณาการเครื่องมือความปลอดภัยจำนวน 4 ชนิดเข้าไปในแต่ละขั้นตอน โดยประกอบด้วย SonarQube (ตรวจสอบโค้ด) Snyk, Trivy (สร้าง) และ OWASP ZAP (ทดสอบ) โดยทำการ Deploy โค้ด 3 ขนาดและ 3 รูปแบบผ่านไปป์ไลน์ทั้งสองแบบ และสังเกตแนวโน้มการใช้ทรัพยากรเวลา หน่วยความจำ และ CPU ซึ่งสามารถสรุปผลการทดลองได้ดังนี้

5.1.1.1 การบูรณาการเครื่องมือรักษาความปลอดภัย 4 ชนิดเข้ากับ CI/CD ไปป์ไลน์ ส่งผลให้เวลาในการ Deploy โดยรวมเพิ่มขึ้นประมาณ 2 เท่า เมื่อเทียบกับไปป์ไลน์แบบธรรมดา

5.1.1.2 ในระหว่างที่ขั้นตอนของเครื่องมือ Trivy เริ่มดำเนินการ ตลอดจนไปถึงขั้นตอนการ Deploy และขั้นตอน OWASP ZAP ทรัพยากรหน่วยความจำมีแนวโน้มที่เพิ่มสูงขึ้น ในขณะที่แนวโน้มการใช้ทรัพยากรหน่วยความจำของไปป์ไลน์แบบธรรมดาอยู่ในระดับที่คงที่ อย่างไรก็ตาม การเพิ่มขึ้นนี้มีสัดส่วนน้อยเมื่อเปรียบเทียบกัน และถือว่าได้เป็นการใช้ทรัพยากรได้อย่างมีประสิทธิภาพมากขึ้น

ผลสรุปของการตรวจสอบการใช้ทรัพยากรของไปป์ไลน์ความปลอดภัยแบบหลายชั้น พบว่าทรัพยากรเวลาที่เพิ่มขึ้นของไปป์ไลน์ความปลอดภัยแบบหลายชั้นไม่ได้ส่งผลกระทบต่อความถี่ในการ Deploy และองค์กรยังคงสามารถ Deploy โค้ดได้หลายครั้งต่อวัน อีกทั้งการเพิ่มขึ้นนี้ถือว่าอยู่ในระดับที่ยอมรับได้ เมื่อพิจารณาถึงประโยชน์ที่ได้รับในด้านความปลอดภัย ซึ่งสามารถนำมาใช้ได้ระยะยาว โดยการเพิ่มเครื่องมือความปลอดภัยในแต่ละขั้นตอนจะช่วยลดความเสี่ยงในการเกิดข้อผิดพลาดในสภาพแวดล้อมการผลิต ซึ่งอาจส่งผลให้ต้องใช้เวลาในการแก้ไขปัญหามากขึ้นในภายหลัง

5.1.2 การตรวจสอบทรัพยากรใน Build Stage

การทดลองนี้ตรวจสอบทรัพยากรก่อนและหลังการแก้ไขช่องโหว่ด้วย Snyk โดยแนวโน้มการใช้ทรัพยากรทั้งหมดไม่ส่งผลกระทบเชิงลบต่อเซิร์ฟเวอร์ สรุปผลการทดลองได้ดังนี้

5.1.2.1 เวลาดำเนินการที่ใช้ใน Build Stage ทั้งก่อนและหลังแก้ไข ไม่ได้มีความแตกต่างกันมาก เนื่องจากจำนวน Dependencies ยังมีจำนวนเท่าเดิม เพียงแต่อัปเดตเวอร์ชัน

5.1.2.2 ทรัพยากรหน่วยความจำที่โค้ดทุกขนาดใช้มีแนวโน้มลดลงเล็กน้อยหลังแก้ไข เนื่องจาก Dependencies หลายรายการได้ถูกปรับปรุงและอัปเดตเป็นเวอร์ชันที่ใหม่กว่า ส่งผลให้ใช้พื้นที่หน่วยความจำน้อยลง อีกทั้งในการทดลองนี้ โค้ดขนาดกลางมีประสิทธิภาพในการอัปเดต Dependencies ได้ดีที่สุด และทรัพยากรหน่วยความจำที่ใช้ลดลงจากก่อนอัปเดตถึงร้อยละ 20 ในขณะที่โค้ดขนาดใหญ่ลดลงเพียงร้อยละ 3.45 เท่านั้น แม้จะน้อยกว่า แต่ก็ยังแสดงให้เห็นอย่างชัดเจนว่าการอัปเดต Dependencies มีผลกระทบที่ดีต่อการลดการใช้หน่วยความจำ

5.1.2.3 ทรัพยากร CPU มีแนวโน้มเพิ่มสูงขึ้นเล็กน้อยหลังแก้ไข เนื่องจากการอัปเดตเป็นเวอร์ชันใหม่ทำให้ต้องมีการประมวลผลใหม่ และ Dependency บางตัวอาจมีความซับซ้อนมากขึ้น

ผลสรุปของการตรวจสอบทรัพยากรใน Build Stage พบว่าทรัพยากรหน่วยความจำที่ลดลงแสดงให้เห็นถึงประสิทธิภาพที่ดีขึ้นของการอัปเดตและปรับปรุง Dependencies ซึ่งเป็นผลมาจากการลดช่องโหว่ และการเพิ่มขึ้นของ CPU แสดงให้เห็นถึงความซับซ้อนที่เพิ่มขึ้นในการประมวลผล Dependencies ที่ถูกอัปเดตใหม่ ซึ่งเป็นการแลกเปลี่ยนทรัพยากรที่ยอมรับได้เพื่อช่องโหว่ที่ลดลงและความปลอดภัยที่เพิ่มขึ้น อีกทั้ง CPU ที่เพิ่มขึ้นไม่ได้ส่งผลกระทบต่อความเร็วของการดำเนินการของไปป์ไลน์แต่อย่างใด และสุดท้าย เวลาดำเนินการที่เปลี่ยนแปลงน้อย แสดงให้เห็นว่าการการลดช่องโหว่ไม่จำเป็นต้องเพิ่มทรัพยากรด้านเวลาเสมอไป โดยรวมแล้ว การลดช่องโหว่ในขั้นตอน Build Stage ผ่านการอัปเดต Dependencies ส่งผลให้เกิดการเปลี่ยนแปลงในการใช้ทรัพยากรระบบ ซึ่งเป็นการแลกเปลี่ยนทรัพยากรที่คุ้มค่าเมื่อพิจารณาถึงความปลอดภัยที่เพิ่มขึ้นของซอฟต์แวร์ และเป็นการเพิ่มความเชื่อมั่นให้แก่องค์กรว่าได้ส่งมอบซอฟต์แวร์ที่ปลอดภัยให้กับลูกค้า

5.1.3 การลดลงของช่องโหว่ใน Build Stage อยู่ที่ร้อยละ 16.5 โดยคำนวณจากค่าเฉลี่ยของช่องโหว่ที่ได้รับการแก้ไขในโค้ดทุกขนาด และการทดลองนี้ได้ลดช่องโหว่ในระดับความรุนแรงระดับวิกฤต (Critical) และระดับสูง (High) เท่านั้น เนื่องจากมีข้อจำกัดด้านทรัพยากรของเซิร์ฟเวอร์ที่การทดลองใช้ นอกจากนี้ หากช่องโหว่ได้รับการแก้ไขในทุกระดับ ก็จะสามารถลดได้สูงมากยิ่งขึ้น

5.2 ข้อเสนอแนะ

5.2.1 การอัปเดต Dependencies ในขั้นตอนสร้างสามารถลดช่องโหว่ได้จริง แต่มีโอกาสที่การ Deploy จะไม่สำเร็จได้ เนื่องจากเวอร์ชัน Dependency อาจขัดแย้งกับโค้ดที่ใช้อยู่ ดังตารางที่ 5-1

แสดงปัญหาการ Deploy หลังการอัปเดตโดยการแก้ไขของทีม DevOps เพียงฝ่ายเดียว ซึ่งแสดงให้เห็นว่ายิ่งมีการอัปเดตได้มีประสิทธิภาพ ก็จะทำให้ส่งผลต่อการ Deploy ล้มเหลว ดังนั้นควรแก้ไขเฉพาะ Dependencies ที่มีความรุนแรงระดับวิกฤตและสูง พร้อมสื่อสารกับทีมพัฒนาเพิ่มเติม

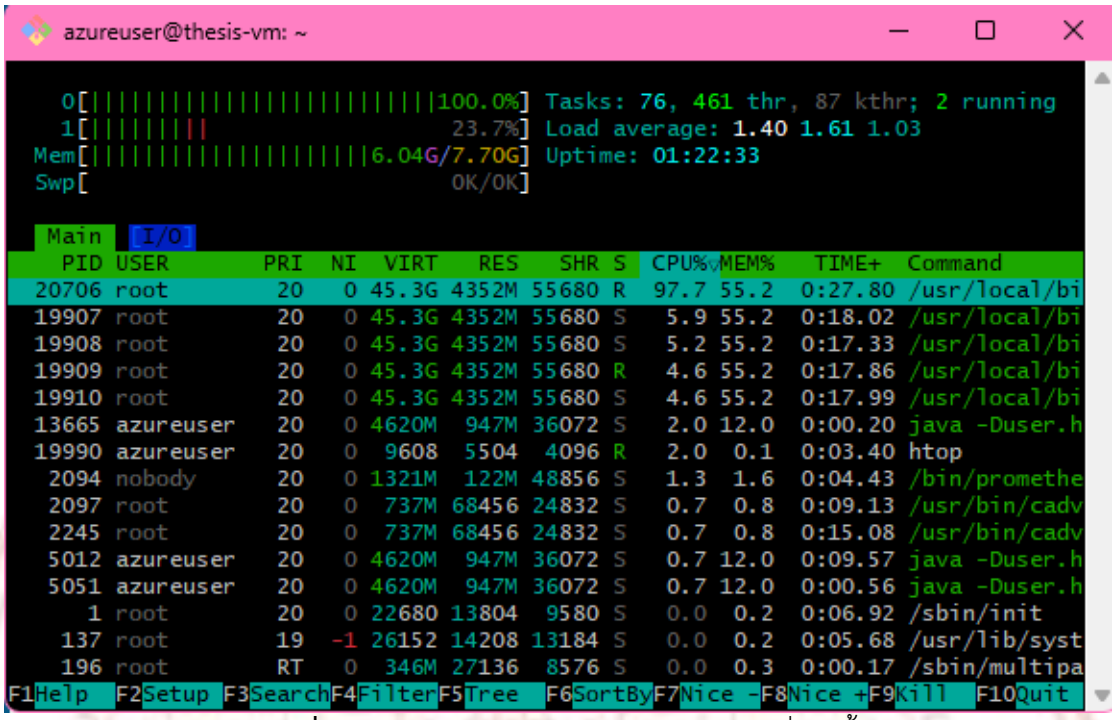
ตารางที่ 5-1 การแสดงปัญหาการ Deploy หลังจากการอัปเดต

ขนาด	จำนวนโค้ด	จำนวนโค้ดที่ Deploy ผ่าน	จำนวนโค้ดที่ Deploy ไม่ผ่าน
เล็ก	3	2	1
กลาง	3	0	3
ใหญ่	3	1	2

5.2.2 CI/CD ไปป์ไลน์เป็นกระบวนการอัตโนมัติและต้องการความเร็ว โดยทรัพยากรหน่วยความจำส่งผลกระทบต่อเวลาได้ชัดเจน อีกทั้งทรัพยากรหน่วยความจำมีผลกระทบอย่างชัดเจนต่อการดำเนินงานของไปป์ไลน์ จากการสังเกตจากการทดลอง พบว่าโดยปกติไปป์ไลน์จะใช้หน่วยความจำสูงในช่วง Build Stage หาก Image มีขนาดใหญ่ และเมื่อไปป์ไลน์ได้ใช้ทรัพยากรหน่วยความจำสูงขึ้นอย่างต่อเนื่องจนใกล้เต็มขนาดหน่วยความจำของเครื่องเสมือนที่กำหนดไว้ จะส่งผลให้ไปป์ไลน์หยุดชะงัก ดังภาพที่ 5-1 และภาพที่ 5-2 จะเป็นหน้าจอแสดงผลการใช้ทรัพยากรของคอมพิวเตอร์เสมือน โดยสังเกตที่การใช้ทรัพยากรหน่วยความจำ (Mem) ที่ใกล้ขีดจำกัด ด้วยการเพิ่มขึ้นอย่างต่อเนื่องนี้ ทำให้เครื่องเสมือนตอบสนองช้าหรือค้างในที่สุด เสมือน ทำให้หน่วยความจำเป็นทรัพยากรที่เหมาะกับการนำมาวัดประสิทธิภาพมากที่สุด นอกจากนี้ Build Stage เป็นส่วนที่สำคัญในเรื่องของการใช้ทรัพยากรหน่วยความจำ ดังนั้น ผู้ใช้ควรประเมินขนาดของไฟล์โค้ดให้เหมาะสมกับสเปคของเครื่องคอมพิวเตอร์เสมือน เพื่อป้องกันปัญหาการใช้หน่วยความจำเกินขนาด

```
#13 6.395 is not maintianed anymore[0;33m. It is thus unlikely that this bug will
#13 6.395 ever be fixed. Add "@babel/plugin-proposal-private-property-in-object" to
#13 6.395 your devDependencies to work around this error. This will make this message
#13 6.395 go away.[0m
#13 6.395
```

ภาพที่ 5-1 Deploy log การหยุดชะงักของไปป์ไลน์ใน Build Stage



ภาพที่ 5-2 การใช้งานทรัพยากรหน่วยความจำเริ่มสูงขึ้น

5.3 ข้อจำกัด

5.3.1 ข้อจำกัดด้านการทดสอบความปลอดภัยของแอปพลิเคชัน

ในการทดลองนี้ เครื่องมือ OWASP ZAP ถูกใช้เพื่อทดสอบความปลอดภัยของแอปพลิเคชันเท่านั้น ซึ่งหมายความว่าไปป์ไลน์ CI/CD ที่มีขั้นตอน OWASP ZAP จะสามารถใช้งานได้เฉพาะกับโค้ดที่เป็นแอปพลิเคชันเท่านั้น สำหรับโค้ดรูปแบบอื่น เช่น Back-end ที่ไม่ได้ถูก Deploy เป็นแอปพลิเคชันที่สามารถเข้าถึงผ่าน HTTP/HTTPS ได้ ทำให้เครื่องมือ OWASP ZAP ไม่สามารถทำการทดสอบได้ เนื่องจากต้องทำการทดสอบโดยการเจาะระบบผ่าน HTTP/HTTPS ซึ่งหากไม่มีแอปพลิเคชันที่สามารถเข้าถึงได้ การทดสอบจะไม่สามารถดำเนินการ และจะไม่สามารถสร้างรายงานผลการทดสอบได้

5.3.2 ข้อจำกัดด้านการเชื่อมต่อฐานข้อมูล

การทดลองนี้จะไม่ได้รวมการเชื่อมต่อกับฐานข้อมูลจริง เนื่องจากความหลากหลายของรูปแบบฐานข้อมูล (Database) ที่อาจใช้ในแต่ละโค้ด และเพื่อให้สอดคล้องกับขอบเขตของงานวิจัยที่เน้นการวิเคราะห์ไฟล์โค้ดเป็นหลัก แต่การเพิ่มการทดสอบกับฐานข้อมูลจริงจะช่วยให้เครื่องมือรักษาความปลอดภัยสามารถตรวจจับและระบุช่องโหว่ได้อย่างครอบคลุมและมีประสิทธิภาพมากยิ่งขึ้น

บรรณานุกรม

1. Aouni, F. E., et al. (2025). “A systematic literature review on Agile, Cloud, and DevOps integration: Challenges, benefits.” Information and Software Technology. Amsterdam : Elsevier, (177: 107569). https://doi.org/10.1016/j.infsof.2024.107569
2. Ironhack. (2025). [online]. DevOps in 2025: Enhancing software delivery and collaboration. [cited 28 Jan. 2025]. Available from : URL : https://medium.com/@ironhack/devops-in-2025-enhancing-software-delivery-and-collaboration-b1b39b99dbe9
3. eSENTIRE. (2024). [online]. Cybercrime to cost the world $9.5 trillion USD annually In 2024. [cited 28 Jan. 2025]. Available from : URL : https://www.esentire.com/web-native-pages/cybercrime-to-cost-the-world-9-5-trillion-usd-annually-in-2024
4. CloudBolt. (2024). [online]. DevSecOps tools for modern enterprises: Enhancing security and productivity. [cited 7 Feb. 2025]. Available from : URL : https://www.cloudbolt.io/data-center-automation/devsecops-tools/
5. TechTarget. (n.d.). [online]. CI/CD pipelines explained: Everything you need to know. [cited 28 Jan. 2025]. Available from : URL : https://www.techtarget.com/searchsoftwarequality/CI-CD-pipelines-explained-Everything-you-need-to-know
6. K, N., et al. (2023). “CI/CD pipeline with vulnerability mitigation.” In 2023 International Conference on Recent Advances in Science and Engineering Technology (ICRASET). IEEE, (1–6). https://doi.org/10.1109/ICRASET59632.2023.10419921
7. Vighe, S. (2024). “Security for continuous integration and continuous deployment pipeline.” International Research Journal of Modernization in Engineering Technology and Science, (6: 2325-2330). https://www.doi.org/10.56726/IRJMETS50676

8. Deploy, O. (2024). [online]. DevOps pipelines: Key components and 6 steps to build yours. [cited 28 Jan. 2025]. Available from : URL : https://octopus.com/devops/ci-cd/devops-pipeline/?form=MG0AV3
9. Bodipudi, A. (2022). "Integrating vulnerability scanning with continuous integration/continuous deployment (CI/CD) pipelines." European Journal of Advances in Engineering and Technology. 9(2), 49-55. http://dx.doi.org/10.13140/RG.2.2.24534.46404
10. Kumar, S. K. R. (2021). An empirical study of containerized CI/CD pipelines for OpenWhisk serverless applications. Master's Thesis. (Informatics), Department of Informatics, The Entrepreneurial University.
11. Faqih, A., et al. (2024). "Empirical analysis of CI/CD tools usage in GitHub Actions Workflows." Journal of Informatics and Web Engineering. 3, 251–261. https://doi.org/10.33093/jiwe.2024.3.2.18.
12. Marandi, M., Bertia, A., & Silas, S. (2023). "Implementing and automating security scanning to a DevSecOps CI/CD pipeline." In Proceedings of the 2023 World Conference on Communication & Computing (WCONF). Raipur, India: IEEE, (1–6). https://doi.org/10.1109/WCONF58270.2023.10235015
13. Bhardwaj, P. (2023). Detecting container vulnerabilities leveraging the CICD pipeline. MSc Research Project, (Cybersecurity), School of Computing, National College of Ireland.
14. Onyenweaku, I., et al. (2021). "A SonarQube static analysis of the Spectral Workbench." International Journal of Natural Science and Reviews. 6, 1–15. http://dx.doi.org/10.28933/ijnsr-2020-12-0605
15. Cloudflare. (n.d.). [online]. What is OWASP? What is the OWASP Top 10?. [cited 17 Feb. 2025]. Available from : URL : https://www.cloudflare.com/learning/security/threats/owasp-top-10/
16. Putta, Y. S. (2023). Enhancing Docker container security. MSc Research Project, School of Computing, National College of Ireland.
17. Bashari Rad, B., Bhatti, H., & Ahmadi, M. (2017). "An introduction to Docker and analysis of its performance." IJCSNS International Journal of Computer Science and Network Security. Vol.17 No.3 : 8-15.

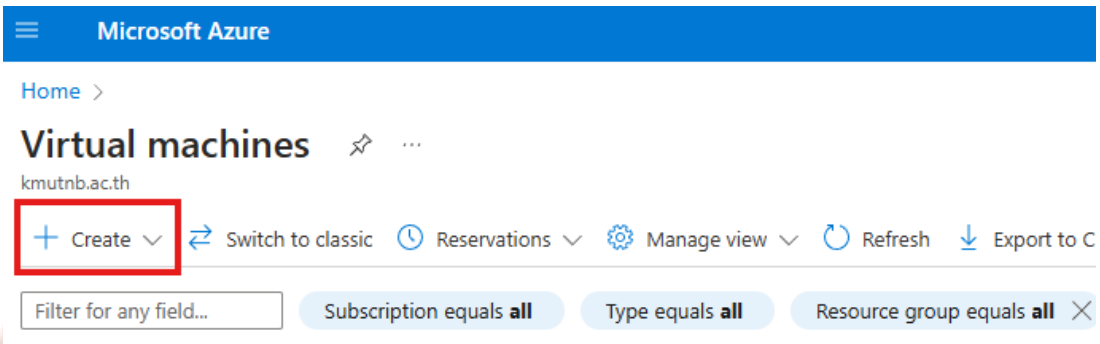
18. Amazon Web Services (AWS). (n.d.). [online]. Security in every stage of CI/CD Pipeline. (Whitepaper). [cited 21 Feb. 2025]. Available from : URL : https://docs.aws.amazon.com/whitepapers/latest/practicing-continuous-integration-continuous-delivery/security-in-every-stage-of-cicd-pipeline.html
19. CircleCI. (n.d.). [online]. Vulnerability management and DevSecOps with CI/CD. (ebook). [cited 24 Jan. 2025]. Available from : URL : https://circleci.com/landing-pages/assets/Vulnerability-Management-ebook_FINAL.pdf
20. Singh, R. (2024). [online]. What is Snyk and use cases of Snyk?. [cited 26 Jan. 2025]. Available from : URL : https://www.devopsschool.com/blog/what-is-snyk-and-use-cases-of-snyk/
21. SA, S. (n.d.). [online]. Analysis scope. [cited 26 Jan. 2025]. Available from : URL : https://docs.sonarsource.com/sonarqube-server/latest/project-administration/analysis-scope/
22. Mahler, C. (2024). [online]. A guide to CI/CD pipeline performance monitoring. [cited 26 Jan. 2025]. Available from : URL : https://www.influxdata.com/blog/guide-ci-cd-pipeline-performance-monitoring/?form=MG0AV3

ภาคผนวก ก

การสร้างคอมพิวเตอร์เสมือนบนไมโครซอฟต์ อาซัวร์

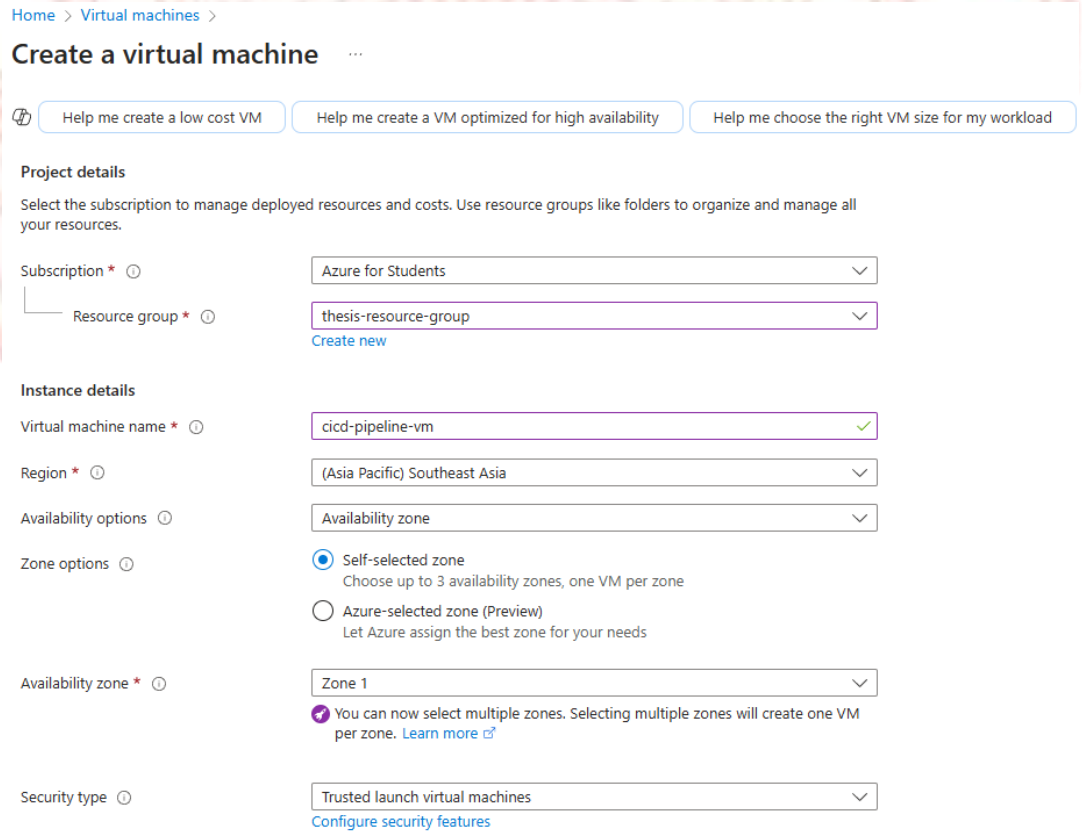
การสร้างคอมพิวเตอร์เสมือนบนไมโครซอฟต์ อาซัวร์

1. สร้างคอมพิวเตอร์เสมือน (Virtual Machine: VM) จำนวนหนึ่งเครื่อง ดังภาพที่ ก-1

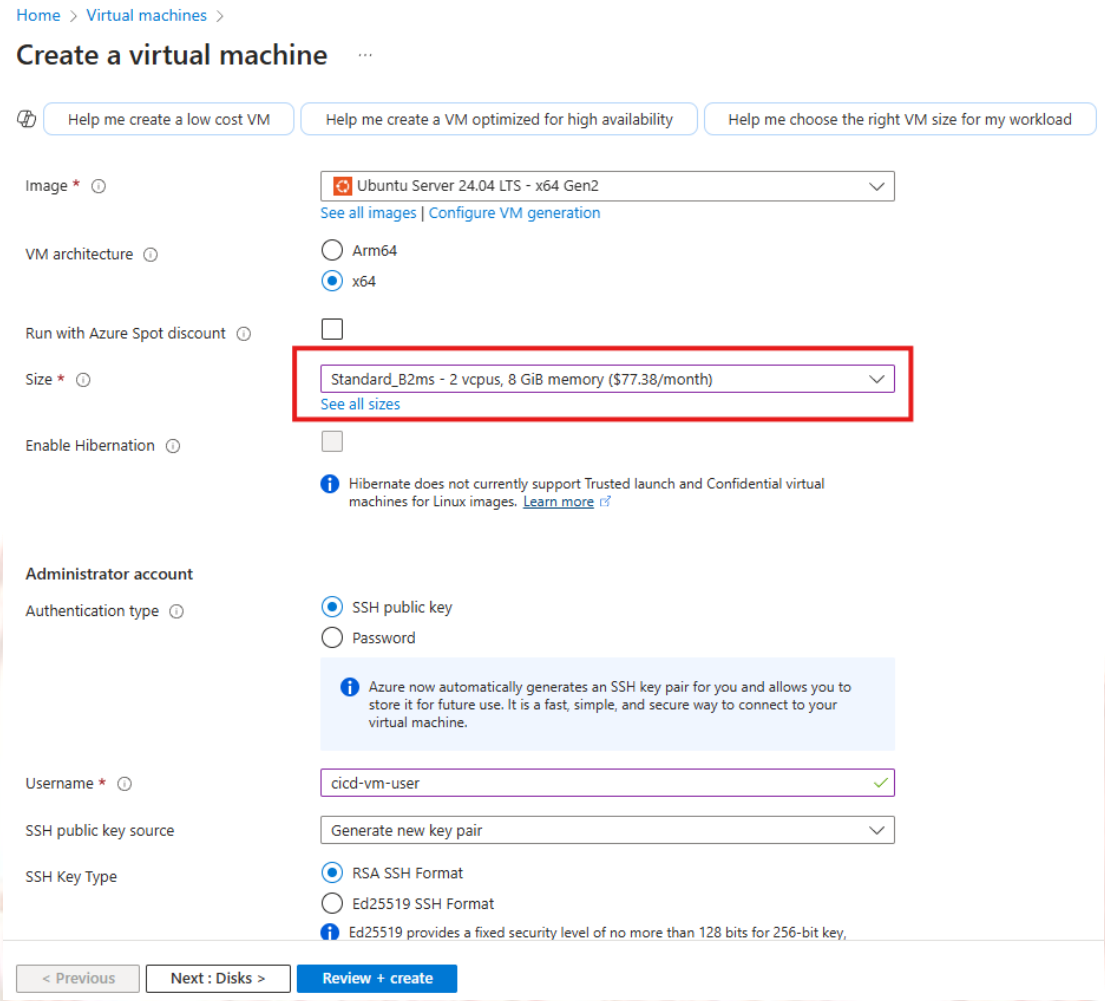


ภาพที่ ก-1 การสร้างคอมพิวเตอร์เสมือน

2. กำหนดชื่อคอมพิวเตอร์เสมือนและตั้งค่าสภาพแวดล้อมของเครื่อง แสดงดังภาพที่ ก-2 และภาพที่ ก-3

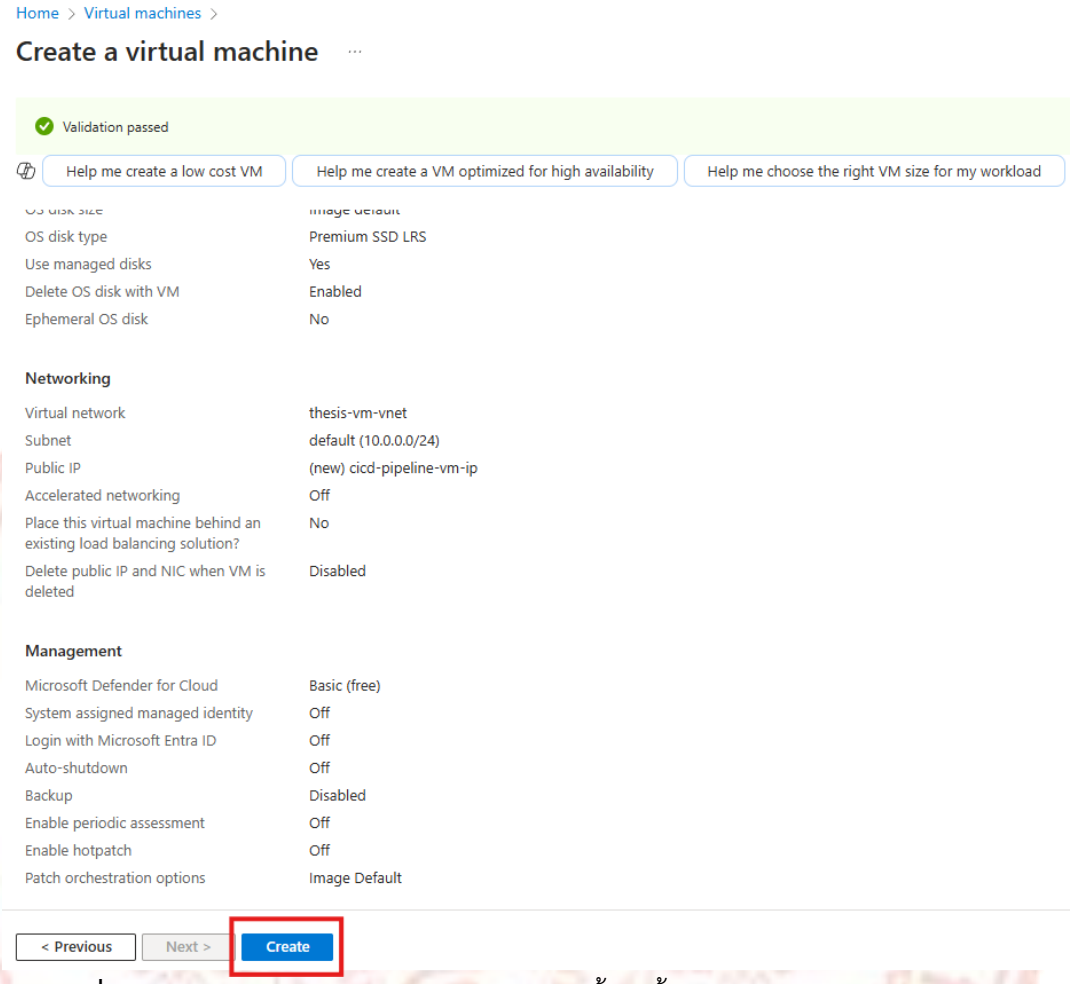


ภาพที่ ก-2 รายละเอียดการสร้างคอมพิวเตอร์เสมือน



ภาพที่ ก-3 การกำหนดขนาดของคอมพิวเตอร์เสมือน

2. ตั้งค่าเครือข่าย (Networking) และรายละเอียดการตั้งค่าทั้งหมด ดังภาพที่ ก-4



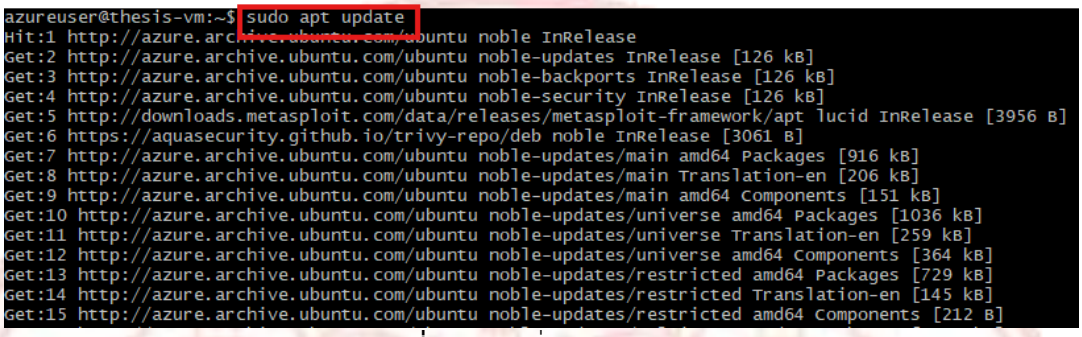
ภาพที่ ก-4 หน้าจอการตรวจสอบรายละเอียดการตั้งค่าทั้งหมดของคอมพิวเตอร์เสมือน

ภาคผนวก ข

การตั้งค่า Docker Container ในคอมพิวเตอร์เสมือน

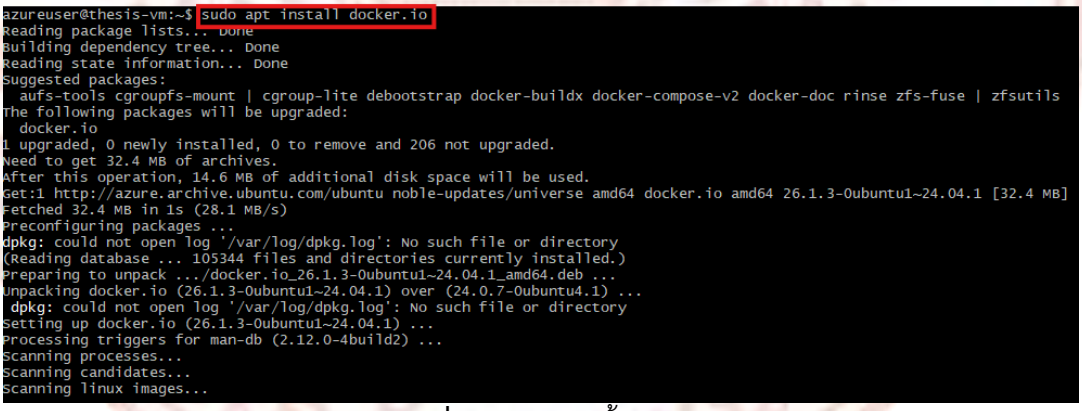
การตั้งค่า Docker Container ในคอมพิวเตอร์เสมือน

1. อัปเดตระบบด้วยคำสั่ง sudo apt update แสดงดังภาพที่ ข-1

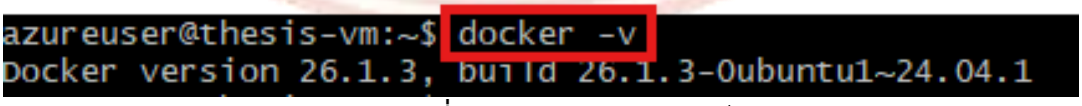


ภาพที่ ข-1 คำสั่งการอัปเดตระบบ

2. ติดตั้ง Docker ด้วยคำสั่ง sudo apt install docker.io ดังภาพที่ ข-2 และตรวจสอบว่า Docker ได้ถูกติดตั้งในเครื่องเรียบร้อยยด้วยคำสั่ง docker -v ดังภาพที่ ข-3



ภาพที่ ข-2 การติดตั้ง Docker



ภาพที่ ข-3 การตรวจสอบเวอร์ชันของ Docker

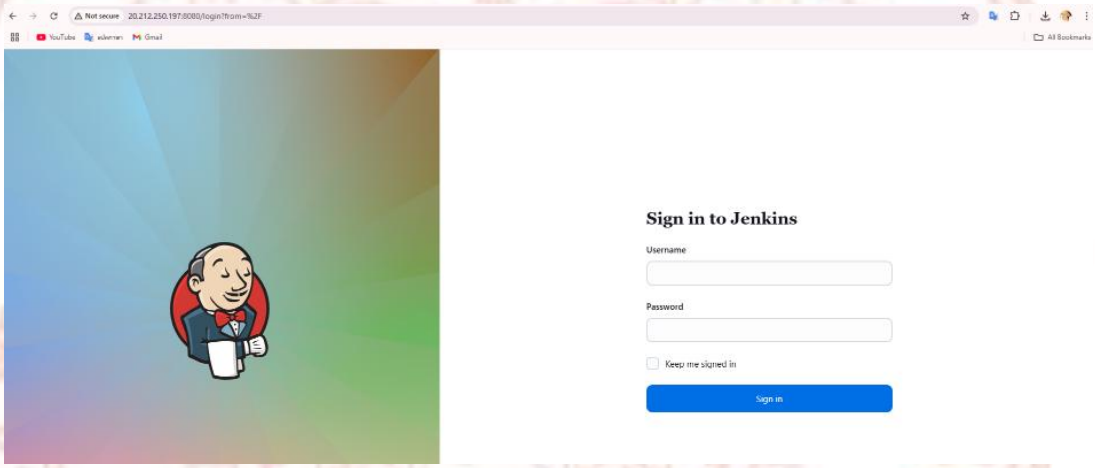
3. ติดตั้ง Jenkins และ Jenkins Controller ด้วยคำสั่งต่อไปนี้

docker run -d --name jenkins-blueocean --network Jenkins

-v jenkins-data:/var/jenkins_home -v jenkins-docker-certs:/certs/client:ro -p 8080:8080 myjenkins-blueocean:2.479.1-1

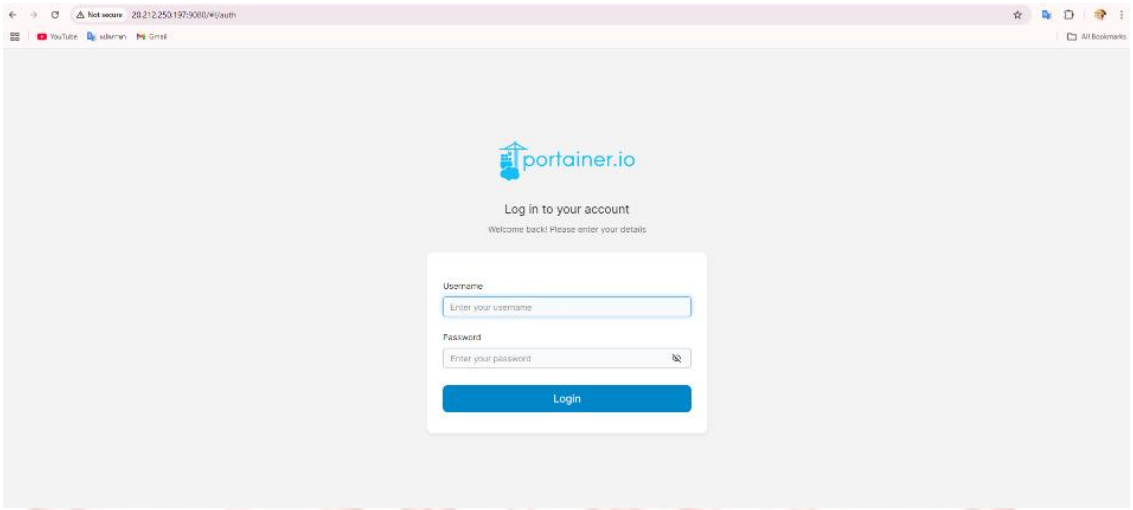
docker run -d --name jenkins-docker --network Jenkins -e DOCKER_TLS_CERTDIR=/certs -v jenkins-docker-certs:/certs/client -v jenkins-data:/var/jenkins_home -p 2376:2376 docker:dind

หลังจากติดตั้ง Jenkins พร้อมตั้งชื่อผู้ใช้และรหัสผ่านสำเร็จ สามารถเปิดใช้งานผ่านเว็บเบราว์เซอร์ (Browser) ได้ แสดงภาพที่ ข-4



ภาพที่ ข-4 ส่วนติดต่อผู้ใช้กราฟิก (GUI) ของ Jenkins

4. ติดตั้ง Portainer ด้วยคำสั่ง docker run -d -p 8000:8000 -p 9080:9000 --name portainer --restart=always -v /var/run/docker.sock:/var/run/docker.sock -v portainer_data:/data --admin-password xxxxxxxxxx portainer/portainer-ce:2.21.4 หลังจากติดตั้ง Portainer สามารถเปิดใช้งานผ่านเว็บเบราว์เซอร์ได้ แสดงภาพที่ ข-5

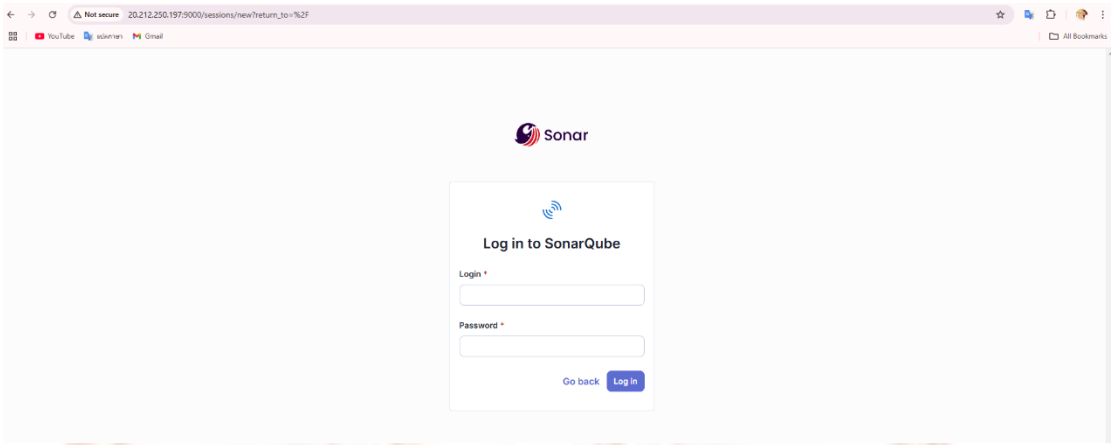


ภาพที่ ข-5 หน้า Portainer ในเว็บเบราว์เซอร์

5. ติดตั้ง Relational Database (Postgres) สำหรับ SonarQube จากนั้นติดตั้ง SonarQube ในรูปแบบ Docker Container ด้วยคำสั่ง docker run -d --name sonarqube –network sonarqube_pg_network \

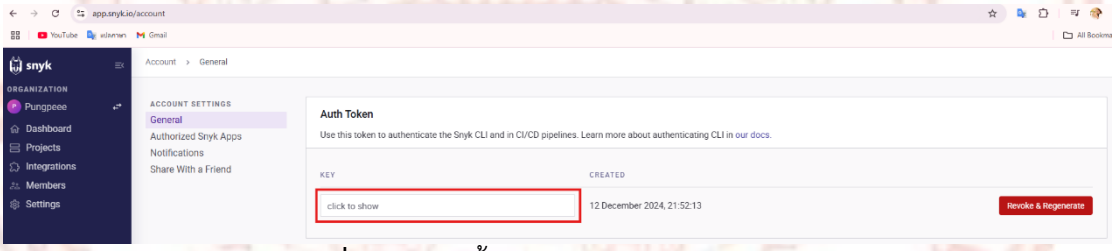
```
  -p 9000:9000 \
  -e SONAR_JDBC_USERNAME=xxxxx \
  -e SONAR_JDBC_PASSWORD=xxxxxxxxx \
  -e SONAR_JDBC_URL=jdbc:postgresql://http://postgres:5432/sonarqube \
  -v sonarqube_data:/opt/sonarqube/data \
  -v sonarqube_extensions:/opt/sonarqube/extensions \
  -v sonarqube_logs:/opt/sonarqube/logs \
  sonarqube:latest
```

หลังจากติดตั้ง SonarQube สามารถเปิดใช้งานผ่านเว็บเบราว์เซอร์ได้ แสดงภาพที่ ข-6

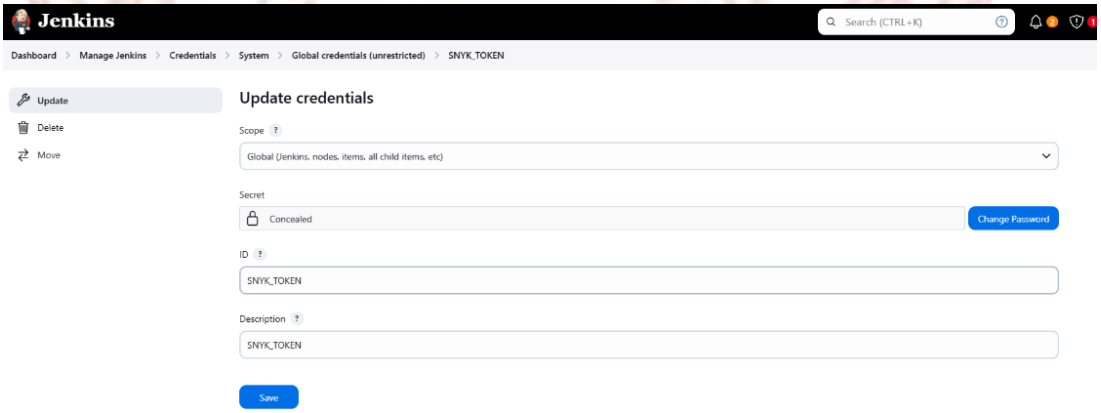


ภาพที่ ข-6 หน้า SonarQube ในเว็บเบราว์เซอร์

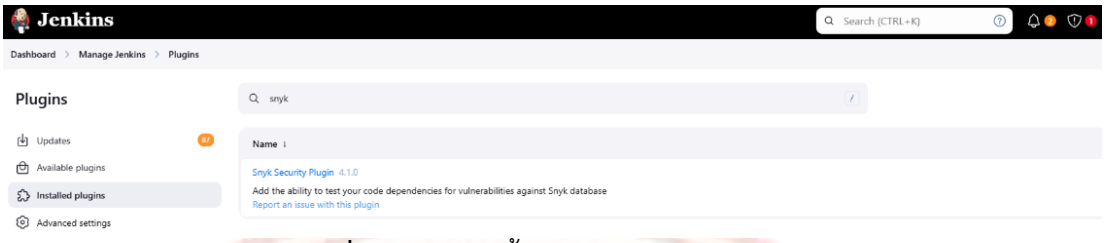
6. การเชื่อมต่อ Snyk กับ Jenkins โดยเข้าสู่เว็บไซต์ https://app.snyk.io/ เพื่อลงทะเบียน เชื่อมต่อกับ GitHub และรับ Snyk โทเค็น (Token) ดังภาพที่ ข-7 จากนั้น นำโทเค็นที่ได้รับไปกำหนดใน Jenkins Credentials ดังภาพที่ ข-8 และติดตั้งปลั๊กอิน (Plugin) "Snyk Security" ใน Jenkins ดังภาพที่ ข-9



ภาพที่ ข-7 หน้าตั้งค่าบัญชีของเว็บ Official Snyk



ภาพที่ ข-8 การกำหนด Snyk Token ลงใน Jenkins Credentials



ภาพที่ ข-9 การติดตั้ง Plugin Snyk ใน Jenkins

7. ติดตั้ง Trivy ใน Container ของ Jenkins (Docker in Docker) ด้วยคำสั่งดังต่อไปนี้

```
sudo apt update
sudo apt install -y wget gnupg lsb-release
wget -qO - https://aquasecurity.github.io/trivy-repo/deb/public.key | sudo gpg --
  dearmor -o /usr/share/keyrings/trivy-keyring.gpg
echo "deb [signed-by=/usr/share/keyrings/trivy-keyring.gpg]
  https://aquasecurity.github.io/trivy-repo/deb $(lsb_release -cs) main" | sudo tee
  /etc/apt/sources.list.d/trivy.list
sudo apt update
sudo apt install -y trivy
```

8. ติดตั้ง OWASP ในรูปแบบ Docker Container ใน Container ของ Jenkins (Docker in Docker) ด้วยคำสั่ง

```
docker run -p 9081:9081 --name devsecops-zap -v /mnt/zap-reports:/zap/wrk -t
  zaproxy/zap-stable:latest
```

9. ลงทะเบียน Docker Hub โดยเข้าสู่เว็บไซต์ https://hub.docker.com/

10. ลงทะเบียน GitHub โดยเข้าสู่เว็บไซต์ https://github.com/

ภาคผนวก ค

การตั้งค่าเครื่องมือ CI/CD

การตั้งค่าเครื่องมือ CI/CD

1. การติดตั้งปลั๊กอินของ Jenkins เพิ่มเติมดังต่อไปนี้

   1.1 SSH Agent

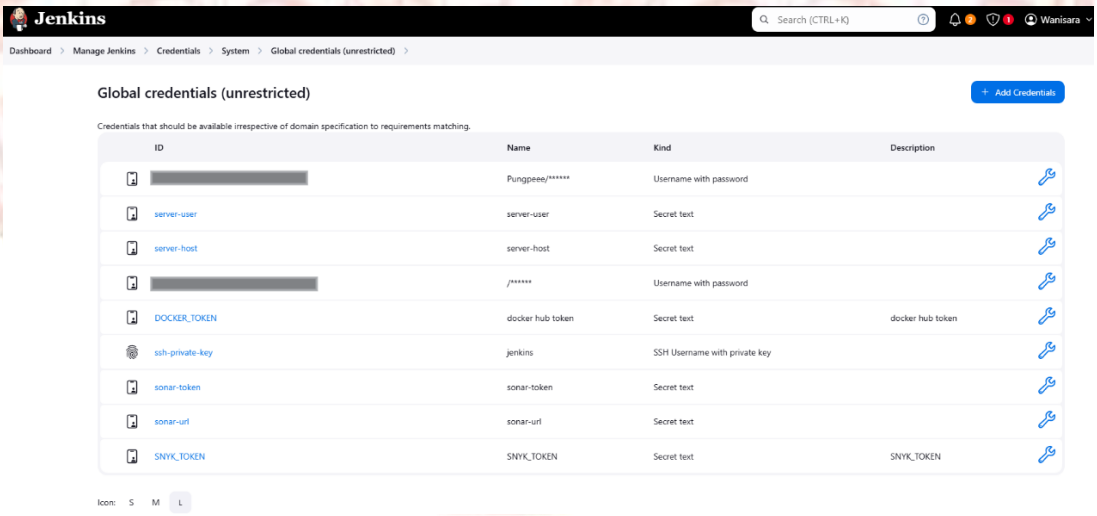
   1.2 SSH server

   1.3 Snyk Security

   1.4 OWASP Markup Formatter

   1.5 Job and Stage Monitoring

2. ในการพัฒนา Jenkins CI/CD ไปป์ไลน์ทั้งสองรูปแบบ (แบบธรรมดาและแบบหลายชั้น) ขั้นตอนแรกคือการกำหนดค่าสภาพแวดล้อมของไปป์ไลน์ เพื่อดึงข้อมูลจาก Jenkins Credentials ที่ได้กำหนดไว้ ดังภาพที่ ค-1 ซึ่งประกอบด้วยค่าที่ใช้ร่วมกัน ได้แก่ SSH KEY ของเครื่องเสมือน ชื่อผู้ใช้เครื่องเสมือน ที่อยู่ IP ของเครื่องเสมือน และ Token จาก Docker Hub การกำหนดค่าในเวิร์คโฟลว์แสดงดังภาพที่ ค-2 สำหรับไปป์ไลน์ความปลอดภัยแบบหลายชั้น จะมีการเพิ่มโทเค็นของเครื่องมือความปลอดภัยบางชนิด ได้แก่ SonarQube และ Snyk ดังภาพที่ ค-3



ภาพที่ ค-1 หน้าการกำหนดค่า Jenkins Credentials

```
environment {
    DOCKER_TOKEN = credentials('DOCKER_TOKEN')
    SSH_PRIVATE_KEY = credentials('ssh-private-key')
    USER = credentials('server-user')
    HOST = credentials('server-host')
}
```

ภาพที่ ค-2 การกำหนดค่าเริ่มต้น Credentials ใน CI/CD Jenkins ทั้งสองเวิร์คโฟลว์

```
environment {
    DOCKER_TOKEN = credentials('DOCKER_TOKEN')
    SONAR_TOKEN = credentials('sonar-token')
    SONAR_HOST_URL = credentials('sonar-url')
    SSH_PRIVATE_KEY = credentials('ssh-private-key')
    USER = credentials('server-user')
    HOST = credentials('server-host')
    SNYK_TOKEN = credentials('SNYK_TOKEN')
}
```

ภาพที่ ค-3 การกำหนดค่าสภาพแวดล้อมเพิ่มเติม Credentials ใน CI/CD Jenkins เวิร์คโฟลว์ที่มีความปลอดภัย

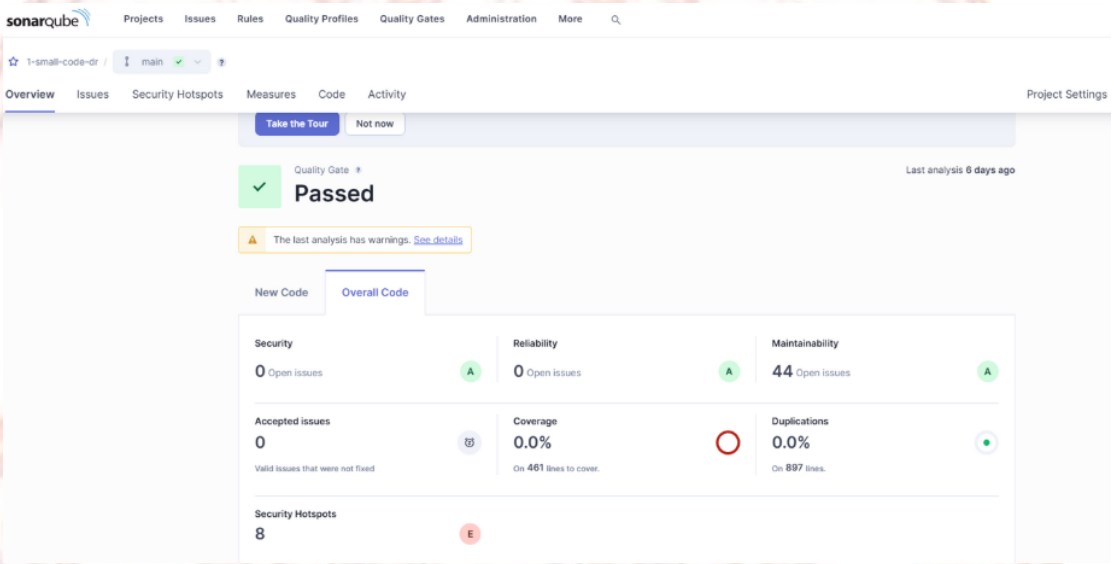
ภาคผนวก ง

รายละเอียดเครื่องมือรักษาความปลอดภัยและเครื่องมืออื่น ๆ ที่เกี่ยวข้อง

รายละเอียดเครื่องมือรักษาความปลอดภัยและเครื่องมืออื่น ๆ ที่เกี่ยวข้อง

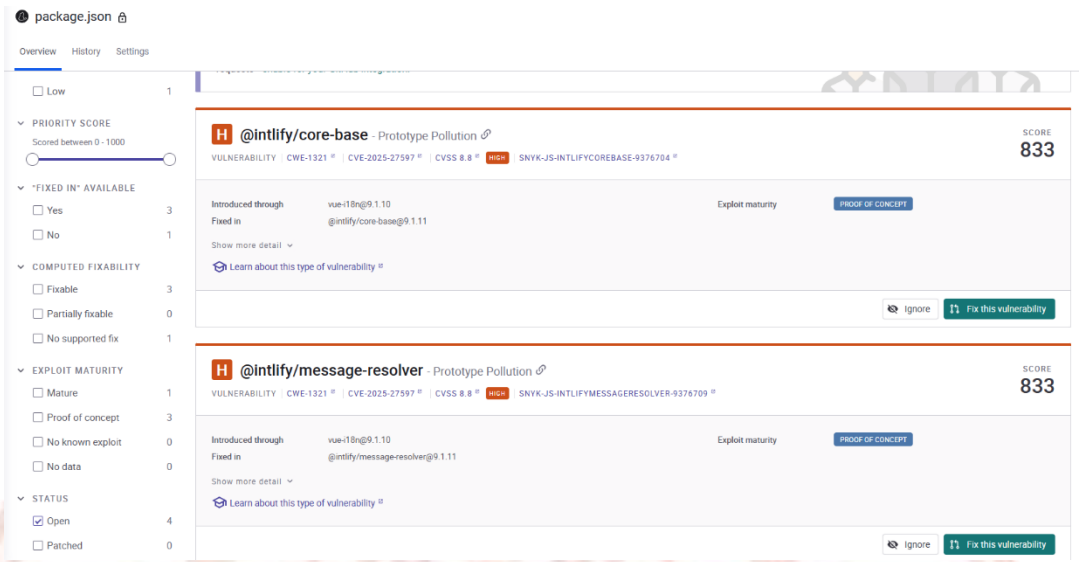
1. การแสดงผลการตรวจสอบความปลอดภัยจากเครื่องมือรักษาความปลอดภัยที่ผสานลงใน CI/CD ไปป์ไลน์แบบปลอดภัย

1.1 SonarQube ที่บูรณาการในไปป์ไลน์ ใช้การเชื่อมต่อผ่านโทเค็นและแสดงรายงานผลการสแกนแยกตามโปรเจกต์ โดยดูผลลัพธ์ได้บนเบราว์เซอร์ผ่าน IP ของเซิร์ฟเวอร์ที่ทำการ Deploy ดังภาพที่ ง-1

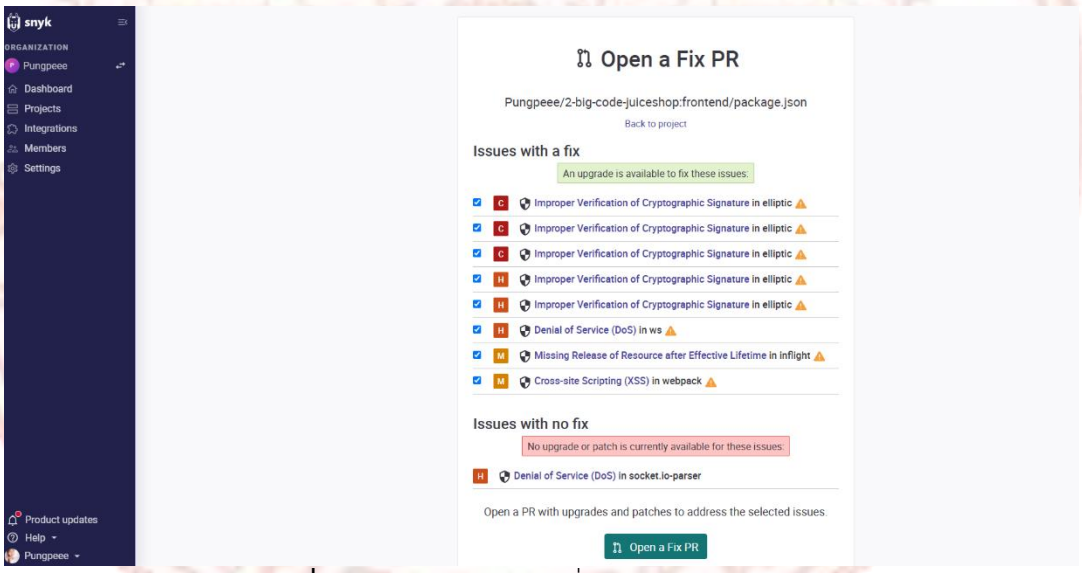


ภาพที่ ง-1 หน้าจอการรายงานผลสแกนของ SonarQube ของโปรเจกต์

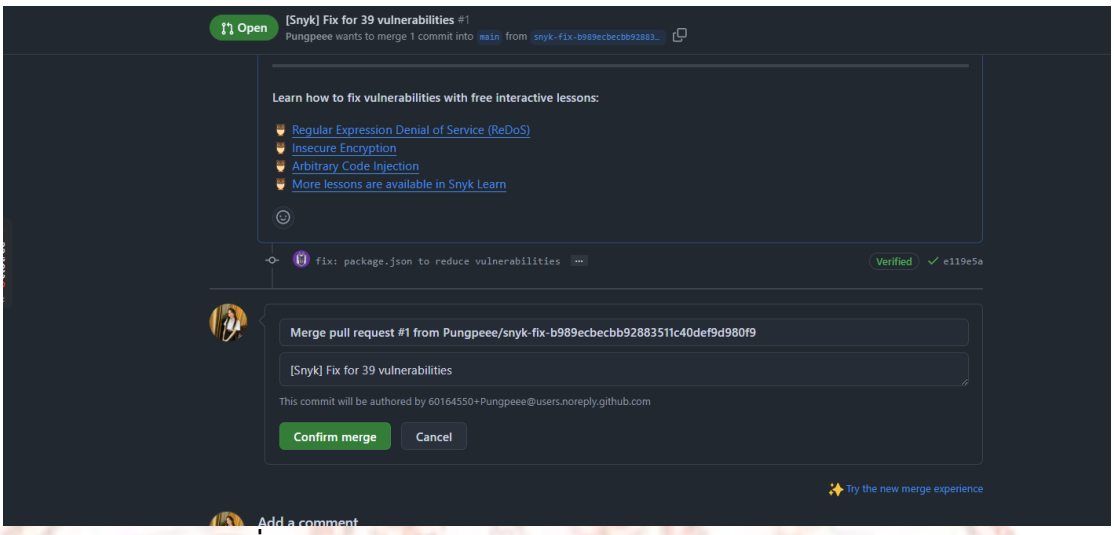
1.2 Snyk ทำงานโดยใช้โทเค็นในการเชื่อมต่อ และแสดงผลลัพธ์ผ่านลิงก์ใน Deploy Log ซึ่งจะนำไปยังหน้า Snyk จากนั้น เมื่อเลือกไฟล์ package.json ที่เป็นที่เก็บของ Dependencies สามารถเลือก Dependencies ที่ต้องการอัปเดตตามคำแนะนำของ Snyk ได้ ดังภาพที่ ง-2 และยืนยันการเปลี่ยนแปลงโค้ด โดยได้สร้าง GitHub Pull Request ตามที่ปรากฏในภาพที่ ง-3 สุดท้ายจะนำไปยังหน้า GitHub โดยอัตโนมัติ ดังภาพที่ ง-4



ภาพที่ ง-2 การเลือก Dependencies ที่ต้องการจะอัปเดตใน Snyk

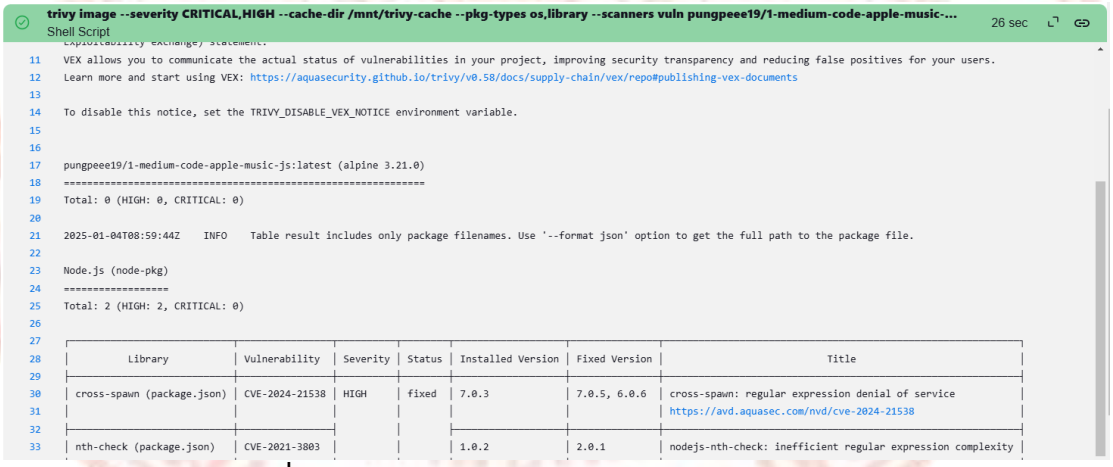


ภาพที่ ง-3 การยืนยันการเปลี่ยนแปลงโค้ดของ Snyk



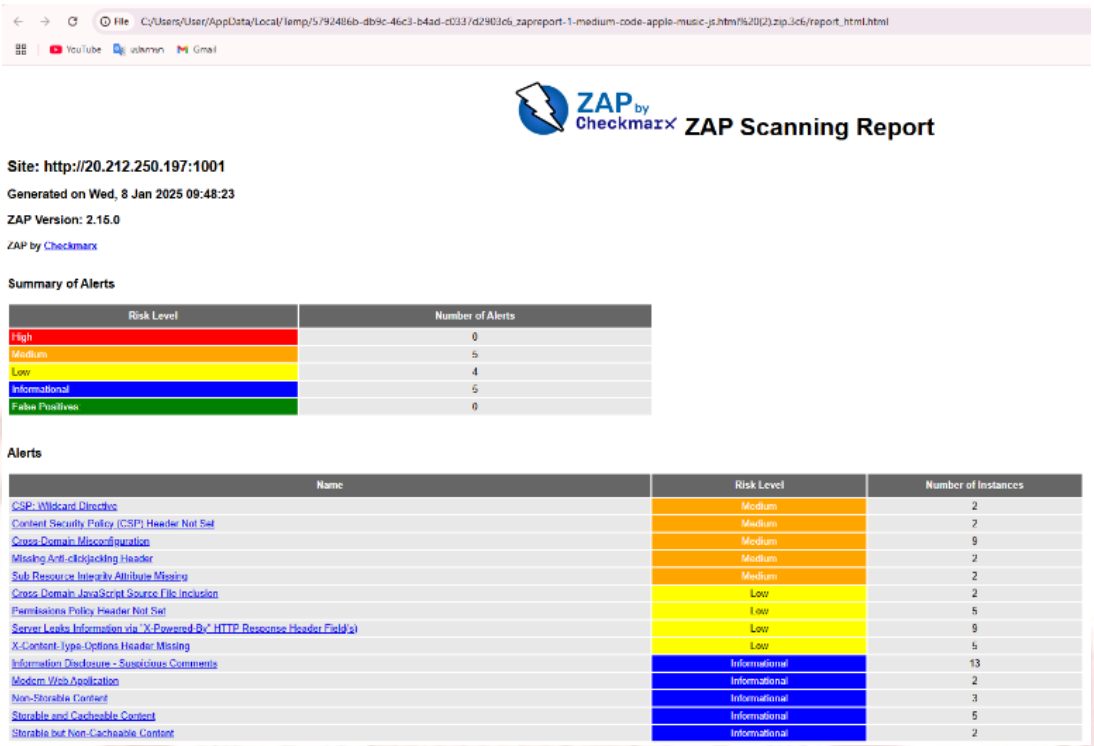
ภาพที่ ง-4 การยืนยันการ Pull Request ของ Snyk บน GitHub

1.3 Trivy จะแสดงผลผ่าน Deploy ดังภาพที่ ง-5



ภาพที่ ง-5 ตัวอย่างผลการรายงานความปลอดภัยของ Trivy

1.4 OWASP ZAP แสดงผลผ่านไฟล์ .HTML ดังภาพที่ ง-6



ZAP by Checkmarx ZAP Scanning Report

Site: http://20.212.250.197:1001

Generated on Wed, 8 Jan 2025 09:48:23

ZAP Version: 2.15.0

ZAP by Checkmarx

Summary of Alerts

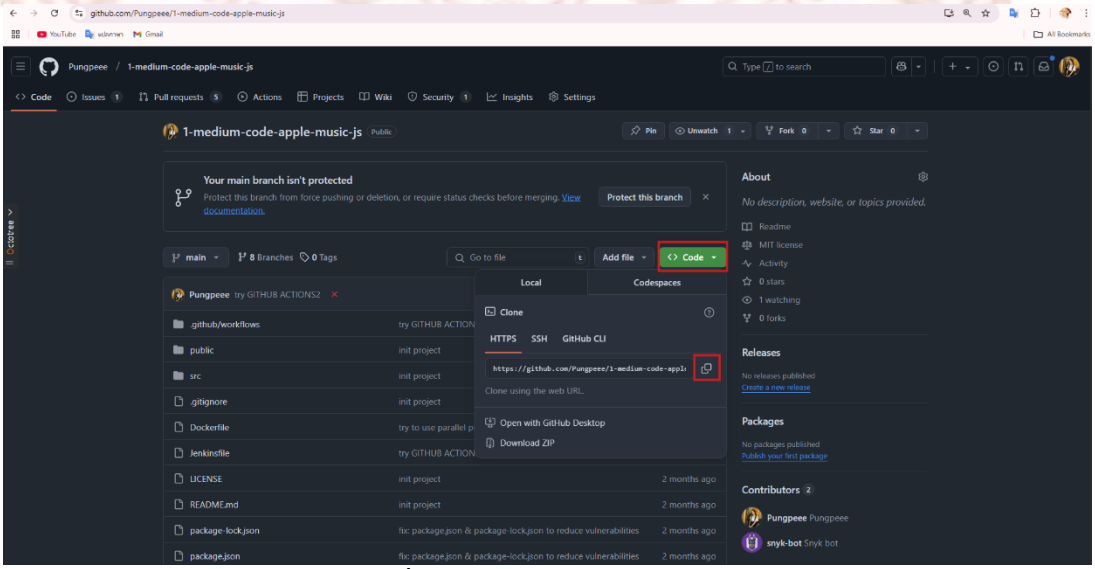
Risk Level	Number of Alerts
High	0
Medium	5
Low	4
Informational	5
False Positives	0

Alerts

Name	Risk Level	Number of Instances
CSP: Wildcard Directive	Medium	2
Content Security Policy (CSP) Header Not Set	Medium	7
Cross-Domain Misconfiguration	Medium	9
Missing Anti-clickjacking Header	Medium	2
Sub Resource Integrity Attribute Missing	Medium	2
Cross-Domain JavaScript Source File Inclusion	Low	2
Permissions Policy Header Not Set	Low	5
Server Leaks Information via "X-Powered-By" HTTP Response Header Field(s)	Low	9
X-Content-Type-Options Header Missing	Low	5
Information Disclosure - Suspicious Comments	Informational	13
Modern Web Application	Informational	2
Non-Storable Content	Informational	3
Storable and Cacheable Content	Informational	5
Storable but Non-Cacheable Content	Informational	2

ภาพที่ ง-6 ตัวอย่างผลการรายงานความปลอดภัยของ OWASP ZAP ในรูปแบบไฟล์ HTML

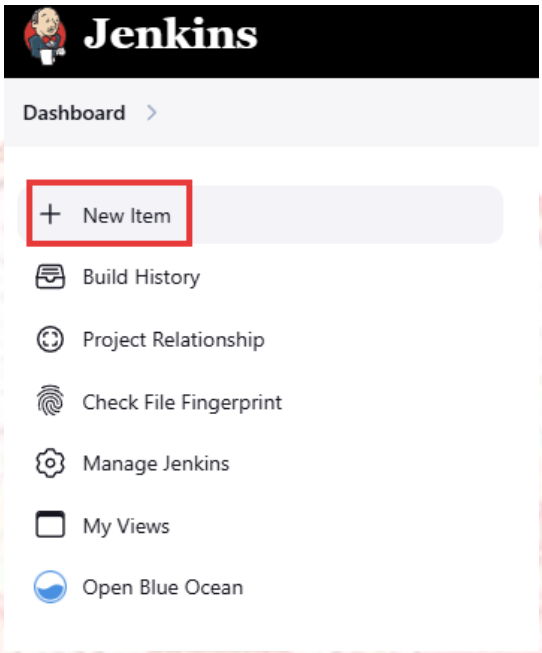
2. การใช้งาน GitHub เมื่อโคลนโปรเจกต์ (Clone Project) หรือสร้างโปรเจกต์ใหม่บน GitHub ให้ดำเนินการคัดลอกลิงก์โปรเจกต์ดังภาพที่ ง-7 เพื่อเชื่อมต่อกับ Jenkins



ภาพที่ ง-7 หน้าโปรเจกต์ของ GitHub

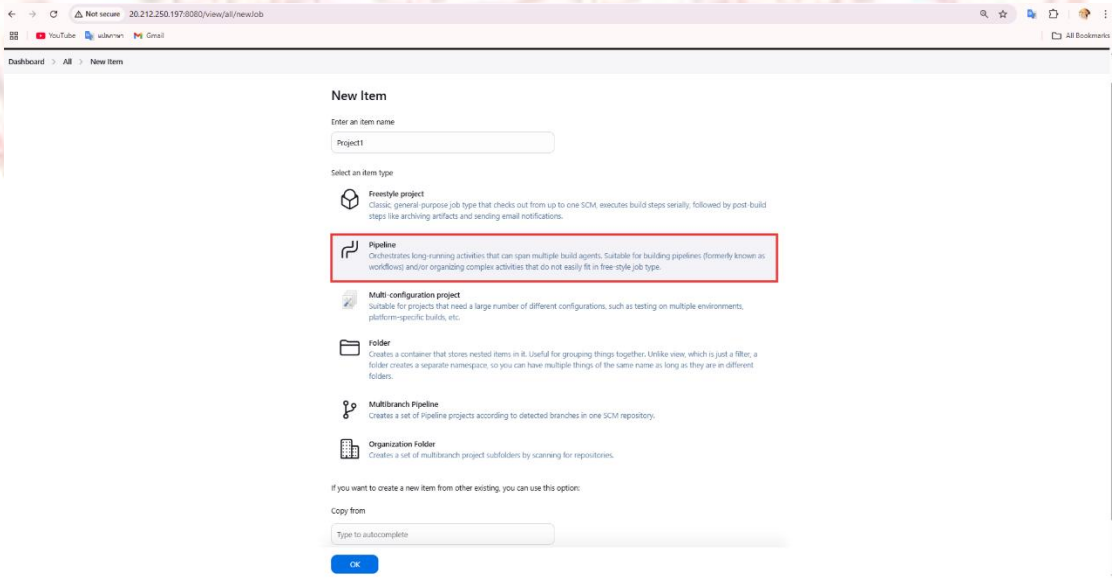
3. การใช้งาน Jenkins เพื่อสร้าง CI/CD ไปป์ไลน์

3.1 เลือกเมนู “New Item” จากมุมซ้ายของหน้า Jenkins ดังภาพที่ ง-8



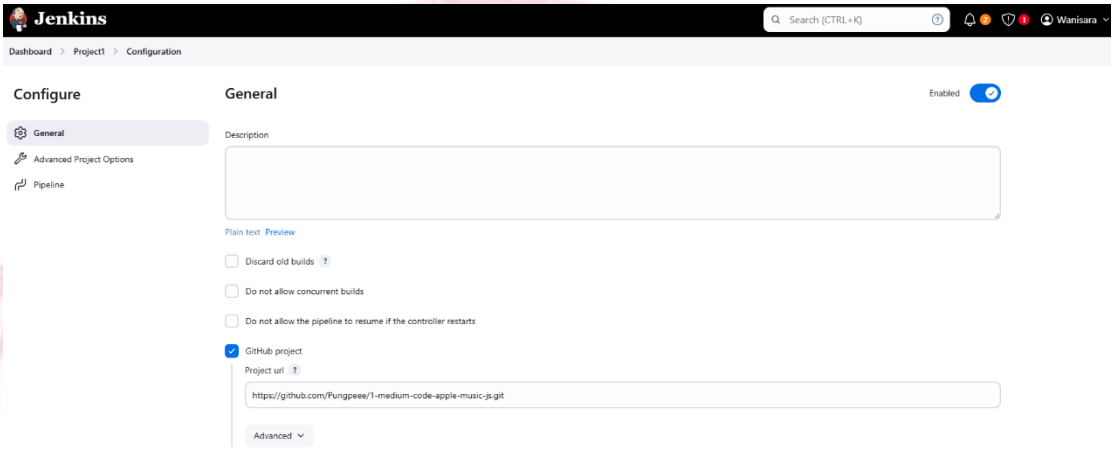
ภาพที่ ง-8 หน้าเมนู New Item เพื่อสร้างไปป์ไลน์ใน Jenkins

3.2 กำหนดชื่อและเลือกชนิดของ Item เป็น Pipeline ดังภาพที่ ง-9

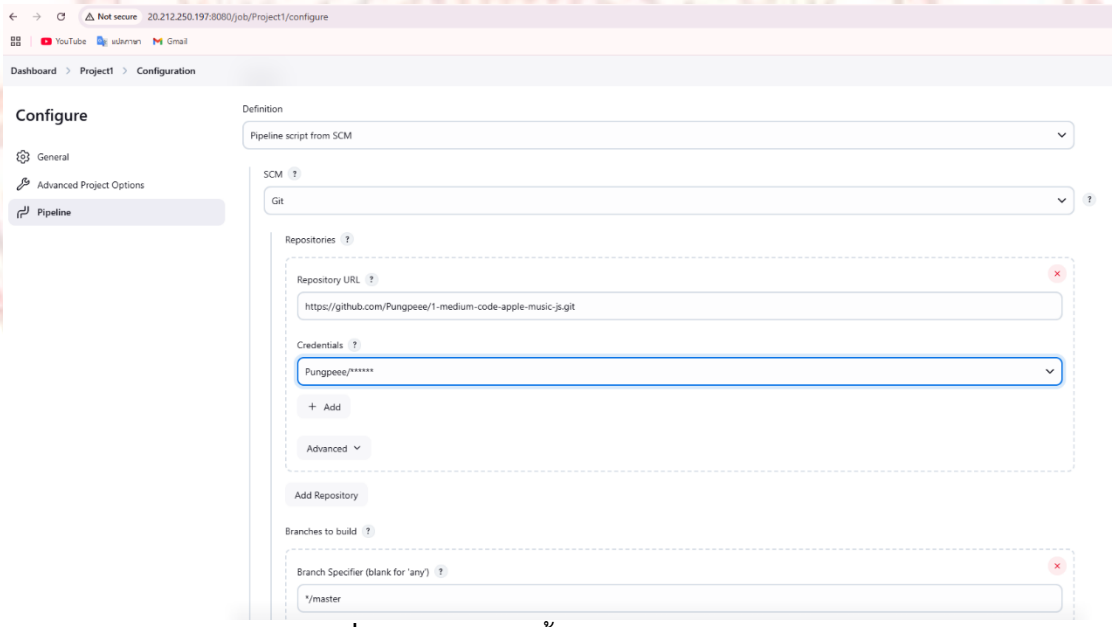


ภาพที่ ง-9 หน้าการตั้งค่าของการสร้างไปป์ไลน์ใน Jenkins

3.3 ในการกำหนดค่า ให้วางลิงก์โปรเจกต์จาก GitHub ลงในช่อง GitHub Project ดังที่แสดงในภาพที่ ง-10 และวางลิงก์ลงใน Repository URL พร้อมเลือก Credentials ที่ตั้งไว้ ดังภาพที่ ง-11



ภาพที่ ง-10 หน้าการตั้งค่าการสร้างไปป์ไลน์ (1)



ภาพที่ ง-11 หน้าการตั้งค่าการสร้างไปป์ไลน์ (2)

ประวัติผู้เขียน

ชื่อ : วณิศรา คำราชา

ชื่อวิทยานิพนธ์ : การเพิ่มประสิทธิภาพ DevSecOps สำหรับความปลอดภัยแบบหลายชั้นใน CI/CD เวิร์คโฟลว์

สาขาวิชา : สาขาวิชาการบริหารเครือข่ายและความมั่นคงปลอดภัยสารสนเทศ

ประวัติ

ประวัติการศึกษา

พ.ศ. 2565 สำเร็จการศึกษาระดับปริญญาตรี วิทยาศาสตรบัณฑิต คณะเทคโนโลยีสารสนเทศ สาขาเทคโนโลยีสารสนเทศ มหาวิทยาลัยเทคโนโลยีพระจอมเกล้าธนบุรี

ประวัติการทำงาน

พ.ศ. 2565 – 2567 ปฏิบัติงานในตำแหน่ง DevOps Engineer บริษัท เวคิน (ประเทศไทย) จำกัด

สถานที่ติดต่อปัจจุบัน

422/387 ซ.สุวินทวงศ์ 34 แขวงแสนแสบ เขตมีนบุรี กรุงเทพมหานคร 10510