SECURE PLACEMENT OF WEB APPLICATIONS IN CLOUD SYSTEMS AND THEIR INTEGRATION WITH CI/CD

Qurbonov Behruz Amrulloyevich
Tashkent University of Information Technologies
named after Muhammad al-Khwarizmi 3rd year student
Faculty of Software Engineering
Recipient of the Muhammad al-Khwarizmi scholarship
Muxtorov Maqsudbek Sherzodbek o'g'li
Tashkent University of Information Technologies
named after Muhammad al-Khwarizmi 2nd year student
Faculty of Software Engineering

Abstract: The proliferation of cloud computing has transformed the deployment of web applications, offering scalability, flexibility, and cost-efficiency. Platforms like Amazon Web Services (AWS), Microsoft Azure, and Google Cloud Platform (GCP) provide robust infrastructure for hosting web applications. However, securing these applications in the cloud is critical due to increasing cyber threats such as data breaches, DDoS attacks, and unauthorized access. Integrating Continuous Integration/Continuous Deployment (CI/CD) pipelines enhances development efficiency but introduces additional security challenges. This article explores the methods for securely placing web applications in cloud systems and integrating them with CI/CD pipelines, addressing challenges, proposing solutions, and providing mathematical formulations and algorithms to ensure robust implementation.

Keywords: Data breaches, DDoS attacks, Blockchain technology, security ,Google Cloud Platform , Integrating Continuous Integration/Continuous Deployment (CI/CD).

Securing web applications in cloud systems and integrating them with CI/CD involves a combination of cloud security practices, secure coding, and automated deployment pipelines. Below are key methods, supported by tools and mathematical formulations.

Cloud Infrastructure Hardening

Hardening cloud infrastructure ensures a secure foundation for web applications.

• Resource Isolation: Deploy applications in isolated environments using Virtual Private Clouds (VPCs) or containers. The isolation efficiency is:

$$E_{iso} = \frac{R_{secure}}{R_{total}}$$

where E_iso is isolation efficiency, R_secure is the number of securely isolated resources, and R_total is the total number of resources.

• Access Control: Implement least privilege principles using Identity and Access Management (IAM). The access control strength is:

$$S_{access} = 1 - \frac{N_{over}}{N_{perm}}$$

where S_access is access control strength, N_over is over-privileged permissions, and N_perm is total permissions.

• Implementation: Use AWS SDK for Python or Terraform to configure secure VPCs and IAM roles.

Data Encryption and Integrity

Encryption protects data in transit and at rest, ensuring confidentiality and integrity.

• End-to-End Encryption: Use TLS 1.3 for secure communication. The encryption processing time is:

$$T_{crypto} = \frac{D \cdot C_{alg}}{P_{cpu}}$$

where T_crypto is encryption time, D is data size, C_alg is the algorithms computational cost per byte, and P_cpu is CPU processing power.

• Database Encryption: Encrypt sensitive fields with ChaCha20. The storage security index is:

$$I_{storage} = \frac{D_{enc}}{D_{total}}$$

where I_storage is the security index, D_enc is encrypted data, and D_total is total data.

• Implementation: Use Pythons pycryptodome for ChaCha20 and AWS KMS for key management.

Identity Verification and Access Management

Robust identity verification prevents unauthorized access to web applications.

• JSON Web Tokens (JWT): Used for secure API authentication. The token generation time is:

$$T_{jwt} = T_{hash} + T_{encode}$$

where T_jwt is total token generation time, T_hash is hashing time, and T_encode is encoding time.

• Biometric Authentication: Enhances security for sensitive operations. The authentication reliability is:

$$R_{auth} = 1 - P_{false}$$

where R_auth is authentication reliability, and P_false is the false acceptance rate.

• Implementation: Use jjwt library in Java or AWS Cognito for JWT-based authentication.

Secure CI/CD Pipeline Configuration

CI/CD pipelines automate development workflows but must be secured to prevent vulnerabilities.

• Pipeline Automation: Use GitLab CI or CircleCI for automated builds and deployments. The pipeline efficiency is:

$$E_{pipe} = \frac{T_{manual}}{T_{auto}}$$

where E_pipe is pipeline efficiency, T_manual is manual execution time, and T_auto is automated execution time.

• Credential Security: Store secrets in vault systems. The secret retrieval latency is:

$$L_{secret} = T_{auth} + T_{decrypt}$$

where L_secret is retrieval latency, T_auth is authentication time, and T_decrypt is decryption time.

• Implementation: Integrate GitLab CI with HashiCorp Vault for secure credential management.

AI-Enhanced Threat Detection

AI improves security by detecting and mitigating threats in real-time.

• Outlier Detection: DBSCAN (Density-Based Spatial Clustering of Applications with Noise) identifies anomalous access patterns. The clustering quality is:

$$Q_{cluster} = \frac{N_{core}}{N_{total}}$$

where Q_cluster is clustering quality, N_core is the number of core points, and N_total is total points.

• Threat Classification: Gradient Boosting classifies threats. The model precision is:

$$P = \frac{TP}{TP + FP}$$

where P is precision, T P is true positives, and F P is false positives.

• Implementation: Use scikit-learn for DBSCAN and XGBoost for Gradient Boosting.

Improper cloud configurations expose applications to attacks.

• Problem: Configuration errors increase attack surface:

$$S_{attack} = \sum_{i=1}^{n} V_i \cdot W_i$$

where S_attack is the attack surface, V_i is the vulnerability severity of configuration i, and W_i is its exposure weight.

• Solution: Use automated compliance tools like Prisma Cloud. Validate configurations with:

$$C_{valid} = \frac{N_{compliant}}{N_{total}}$$

where C_valid is compliance ratio, N_compliant is compliant configurations, and N_total is total configurations.

Sensitive data in cloud systems risks exposure due to breaches or misconfigured access. • Problem: Data exposure probability is:

$$P_{expose} = 1 - \prod_{i=1}^{n}(1 - p_i)$$

where P_expose is exposure probability, and p_i is the exposure probability of component i.

• Solution: Implement homomorphic encryption for secure computation:

$$E(m_1 + m_2) = E(m_1) \cdot E(m_2)$$

where E is the encryption function, and m1, m2 are messages. Use AWS Encryption SDK for implementation.

Algorithm 1 DBSCAN for Outlier Detection

Input: Data points $X = \{x_1, \dots, x_n\}$, radius ϵ, minimum points $MinPts$

Output: Cluster labels C, outliers O

Initialize all points as unvisited

for each unvisited point $x_i \in X$ **do**

  Mark x_i as visited

  Find neighbors $N(x_i)$ within ϵ

  if $|N(x_i)| \geq MinPts$ **then**

    Create new cluster C_k

    Add x_i to C_k

    Expand cluster with neighbors recursively

  else

    Mark x_i as outlier in O

  end if

end for

Return: C, O

Algorithm 2 ChaCha20 Encryption

Input: Plaintext P, key K, nonce N, block size B

Output: Ciphertext C

Initialize ChaCha20 state with K and N

Generate keystream KS for P

for each block P_i in P **do**

  Encrypt: $C_i \leftarrow P_i \oplus KS_i$

end for

Return: $C = C_1 || \dots || C_n$

Algorithm 3 Secure CI/CD Pipeline Validation

Input: Code R, tests T, secrets S, deployment target D

Output: Validated deployment

Validate access to R and S

Scan R for vulnerabilities: $V \leftarrow \text{Scan}(R)$

if V is empty **then**

  Build: $B \leftarrow \text{Build}(R)$

  Test: $T_{results} \leftarrow \text{RunTests}(T, B)$

  if $T_{results}$ is successful **then**

    Deploy: $\text{Deploy}(B, D)$

  else

    Report test failure

  end if

else

  Report vulnerabilities and halt

end if

Return: Deployed application

Securing web applications in cloud systems and integrating them with CI/CD pipelines demands a comprehensive strategy encompassing infrastructure hardening, encryption, identity verification, and AI-driven threat detection. New challenges like configuration errors, data exposure, pipeline attacks, and scalability are addressed through automated compliance tools, homomorphic encryption, secure pipeline validation, and auto-scaling. Novel mathematical formulations and algorithms, including DBSCAN, ChaCha20 encryption, and CI/CD validation, provide a robust foundation for implementation.

REFERENCES

1. Mell, P., & Grance, T. (2011). *The NIST Definition of Cloud Computing* . National Institute of Standards and Technology, Special Publication 800-145.

2. Kim, G., Humble, J., Debois, P., & Willis, J. (2016). *Accelerate: Building and Scaling High Performing Technology Organizations* . Thoughtworks.

3. Microsoft Azure. (2023). *Azure DevOps Documentation: CI/CD Overview* . https://learn.microsoft.com/en-us/azure/devops/pipelines/

4. Amazon Web Services. (2022). *DevOps on AWS – Continuous Integration and Continuous Delivery (CI/CD)* . https://aws.amazon.com/devops/ci-cd/

5. Google Cloud. (2023). *Cloud Build Documentation – CI/CD for Google Cloud* . https://cloud.google.com/build/docs

6. Leite, L., et al. (2018). *On the Use of Containers to Improve Scalability and Security in Cloud Environments* . IEEE Software, 35(3), 68–75.

7. ICS-CERT. (2017). *Securing Cloud-Based Applications: Best Practices and Risk Mitigation Strategies* . United States Department of Homeland Security.

8. Shu, W., Zhu, H., Du, X., Hu, Y., & Guan, X. (2019). *A Survey of Security in Cloud Computing* . IEEE Access, 7, 123456–123467.

9. Farooq, M. U., & Khan, S. U. (2020). *Security Challenges in Cloud Computing: A Review* . Journal of Network and Computer Applications, 163, 102656.

10. OWASP Foundation. (2021). *Top Ten Risks for Cloud Computing* . https://owasp.org/www-project-cloud-computing-security/