International Journal of Computer Engineering and Technology (IJCET)
Volume 14, Issue 1, January-April 2023, pp. 121-147, Article ID: IJCET_14_01_013
Available online at https://iaeme.com/Home/issue/IJCET?Volume=14&Issue=1
ISSN Print: 0976-6367; ISSN Online: 0976-6375; Journal ID: 5751-5249
Impact Factor (2023): 18.14 (Based on Google Scholar Citation)
DOI: https://doi.org/10.34218/IJCET_14_01_013

SECURE CREDENTIAL HANDLING FOR TEST AUTOMATION: A DEEP DIVE INTO VAULT AND CLOUD-NATIVE SECRETS MANAGERS

Pradeepkumar Palanisamy
Anna University, India.

ABSTRACT

In an era defined by persistent cyber threats and stringent regulatory mandates, the security of software development artifacts is paramount. This article examines the critical importance of secure credential handling within automated test environments, which frequently interact with sensitive external services. Specifically, it details the integration of enterprise-grade secrets management solutions—HashiCorp Vault, AWS Secrets Manager, and Azure Key Vault—as a strategic imperative to eliminate the egregious practice of hardcoding sensitive API keys, database passwords, access tokens, and other authentication materials directly into test scripts or their associated configuration files. Beyond mitigating immediate security vulnerabilities, these sophisticated integrations critically foster alignment with contemporary DevSecOps principles by embedding security earlier in the lifecycle, streamline the complex processes of credential lifecycle management (including rotation and revocation), and establish comprehensive, immutable audit trails. This holistic approach thereby significantly fortifies the overall security posture and compliance adherence of the entire software delivery pipeline, from development to deployment.

Keywords: DevSecOps, Test Automation, Secrets Management, HashiCorp Vault, AWS Secrets Manager, Azure Key Vault, API Security, Credential Management, Hardcoding, Audit Trails, Cybersecurity, Compliance, CI/CD Security, Ephemeral Credentials, Least Privilege.

Cite this Article: Pradeepkumar Palanisamy. (2023). Secure Credential Handling for Test Automation: A Deep Dive into Vault and Cloud-Native Secrets Managers. *International Journal of Computer Engineering and Technology (IJCET)*, 14(1), 121-147.

https://iaeme.com/MasterAdmin/Journal_uploads/IJCET/VOLUME_14_ISSUE_1/IJCET_14_01_013.pdf

1. Introduction: The Unseen Vulnerability in Test Automation

Automated testing is the indispensable bedrock of modern software development methodologies, driving rapid feedback loops, accelerating release cycles, and dramatically improving product quality. However, the inherent power of test automation often hinges on its ubiquitous ability to interact seamlessly with a multitude of external systems and resources, including critical application programming interfaces (APIs), production and staging databases, diverse microservices, third-party platform integrations, and cloud infrastructure. These critical interactions invariably demand **credentials** – be it user accounts, access tokens, API keys, or certificates. Historically, and regrettably frequently, a common expedient to facilitate these interactions has been to embed these sensitive authentication materials directly into test scripts, hardcode them within configuration files, or even inadvertently commit them into version control systems. This seemingly convenient practice, while perhaps saving minimal effort in the short term, introduces **profound and often overlooked security vulnerabilities** that remain latent until a malicious actor discovers them or an accidental exposure occurs.

The systemic implications of hardcoded credentials are far-reaching and pose severe risks:

- **Direct Exposure Upon Compromise:** A single successful breach of a test environment, a developer workstation, or even an inadequately secured CI/CD agent can instantly expose all embedded secrets. This single point of failure can lead to cascading compromises across multiple linked systems that share these credentials. For instance, a compromised test API key could potentially grant access to production data or allow a malicious actor to inject harmful data.

- **Irreversible Version Control Leakage:** The act of committing plaintext or easily decipherable credentials into source code repositories, particularly those that are publicly accessible or inadequately protected, creates an indelible record. Even if quickly removed, the credential's history typically remains in the repository's commit history, making it discoverable through forensic tools and potentially accessible indefinitely to those with historical repository access. This type of leakage can lead to widespread and persistent credential compromise, requiring costly and urgent secret rotation across multiple systems.
- **Egregious Compliance Failures:** Hardcoding sensitive data directly violates fundamental tenets of numerous industry regulations and compliance frameworks, such as the General Data Protection Regulation (GDPR), Health Insurance Portability and Accountability Act (HIPAA), Service Organization Control 2 (SOC 2), and Payment Card Industry Data Security Standard (PCI DSS). These mandates explicitly require secure handling, storage, and access control for sensitive information. Non-compliance can result in substantial financial penalties, severe legal repercussions, and significant reputational damage, impacting customer trust and market standing.
- **Prohibitive Operational Burden and Risk:** Credential rotation, a fundamental and critical security practice for mitigating the impact of potential compromises, becomes an extraordinarily labor-intensive, error-prone, and slow task when secrets are haphazardly scattered across numerous test scripts, environments, and configuration files. This manual burden often leads to infrequent or neglected rotations, leaving long-lived credentials vulnerable to brute-force attacks or discovery. The sheer volume of manual updates can also introduce new errors or inconsistencies.
- **Erosion of Trust and Security Posture:** The pervasive presence of hardcoded secrets signals a lax security culture and fundamentally diminishes confidence in the overall security integrity of the testing and deployment pipeline. It indicates a lack of robust security practices embedded throughout the software development lifecycle, which can have downstream impacts on customer trust and investor confidence. It also creates a "security debt" that can be expensive and time-consuming to rectify later.

This article unequivocally posits that robust and centralized **secrets management** is not merely an optional "good-to-have" feature but rather a critical and non-negotiable component of a mature and resilient **DevSecOps strategy** for test automation. We will delve into dedicated, enterprise-grade solutions that provide secure, centralized, and auditable means of managing

credentials, thereby transforming what has traditionally been a significant security blind spot into a foundational pillar of organizational cybersecurity.

2. The DevSecOps Imperative for Secrets Management

The **DevSecOps** philosophy advocates for the proactive integration of security practices throughout the entire software development lifecycle ("shifting security left"). In the specific context of automated testing, this paradigm shift means moving beyond reactive, post-facto security audits or vulnerability scans to proactively designing and implementing secure mechanisms for handling sensitive data from the outset. Security becomes an inherent quality baked into the testing process, rather than an afterthought, ensuring that security considerations are embedded at every stage of the software delivery pipeline.

A mature DevSecOps approach to test automation emphatically demands adherence to several core security principles, creating a more secure and efficient testing environment:

- **Least Privilege Principle (PoLP):** This fundamental security tenet dictates that every entity—whether it's a test environment, a CI/CD pipeline, a test runner, or a specific test case—should only be granted the absolute minimum level of access and the precise permissions necessary to perform its intended function, and critically, only for the duration required. For credential handling, this means a test script should only be able to retrieve the specific API key it needs for a particular test, not access to every secret in the system. If a test runner only needs to read a database password, it should not have permissions to modify or delete that password, nor should it have access to unrelated secrets like SSH keys or production API keys. This significantly reduces the potential blast radius if a component is compromised.
- **Centralized Control and Management:** Establishing a single, authoritative source of truth for all secrets ensures consistency, reduces duplication, and simplifies management. This centralized repository allows for consistent policy enforcement across all environments, from development to production, and provides a clear overview of all credentials in use. It moves away from scattered, unmanaged secrets that are difficult to track or revoke. A centralized system streamlines updates, ensures version control of secrets, and makes it easier to enforce naming conventions and access patterns.
- **Automated Lifecycle Management:** The ability to automatically rotate, revoke, and manage secret lifecycles without requiring manual intervention is paramount. This

includes setting expiry dates for credentials, triggering automated rotations for long-lived secrets (e.g., database passwords for test environments, frequently changing API keys for third-party services), and instantly revoking compromised or unused secrets. Automation significantly reduces the human error factor, eliminates the burden of manual rotations, and ensures continuous adherence to security policies, dramatically improving the overall security hygiene.

- **Comprehensive Auditability and Traceability:** Every single access attempt, modification, creation, or deletion of a secret must be meticulously logged with rich contextual information. This includes details such as who accessed the secret (e.g., specific user, service account, or CI/CD job ID), what secret was accessed, when the access occurred, and from where (e.g., IP address, hostname, unique identifier of the requesting resource). This immutable audit trail is indispensable for security monitoring, detecting anomalous behavior, facilitating rapid incident response in the event of a breach, demonstrating compliance with regulatory requirements, and providing a verifiable record for post-incident forensic analysis.
- **Segregation of Duties (SoD) and Responsibilities:** Implementing clear separation between individuals or systems that define access policies for secrets and those that actually consume or retrieve the secrets is crucial. For instance, the security or operations team might be responsible for configuring and managing the secrets manager itself and defining the overarching access policies. Development or test engineers, on the other hand, are only granted permissions to retrieve secrets through an authorized, automated method, without direct knowledge or manual access to the secret values themselves. This minimizes the risk of insider threats, reduces the potential impact of a single compromised account, and reinforces accountability.

Traditional, ad-hoc credential handling methods fundamentally contradict these critical principles, creating inherent security gaps and operational inefficiencies. Dedicated secrets management solutions, by contrast, are engineered precisely to address these DevSecOps tenets, offering a transformative paradigm shift from insecure, manual practices to systematic, secure, and auditable credential management. They enable security to be an enabler, not a blocker, for agile development and robust testing, ultimately accelerating the delivery of secure software.

3. Core Principles of Secure Credential Handling for Tests

Before delving into the architectural specifics of individual solutions, it is imperative to thoroughly understand the foundational principles that guide secure credential management within the demanding context of test automation. Adherence to these principles is non-negotiable for building a truly secure testing environment:

- **Never Hardcode (The Golden Rule):** This is the absolute, inviolable principle. Under no circumstances should sensitive credentials, such as API keys, database connection strings, passwords, or authentication tokens, be embedded in plaintext within source code files, configuration files (e.g., .properties, .yaml), environment variables in source control, or any easily decipherable format. This practice is the root cause of many credential leaks and must be strictly forbidden through clear organizational policy, enforced by automated code scanning tools, and reinforced through a strong security-aware culture. Any discovery of hardcoded secrets should trigger immediate and critical security alerts.
- **Ephemeral Secrets (Dynamic Secrets) and Short-Lived Credentials:** Whenever technically feasible, the preference should be for **dynamic generation of temporary credentials** rather than the long-term storage of static ones. These ephemeral secrets are provisioned on-demand by the secrets manager (e.g., a unique, time-limited database user and password for a specific test run) and automatically expire after a very short duration (e.g., minutes or hours). This dramatically minimizes the window of exposure if a secret were to be accidentally copied or momentarily exposed. For static secrets that cannot be dynamically generated (e.g., a long-lived third-party API key), they must be short-lived by design and subject to frequent, automated rotation to reduce their utility if compromised.
- **Just-in-Time (JIT) Access and In-Memory Handling:** Secrets should be retrieved from the secrets manager only at the precise moment they are required by a test script. They should then be held exclusively in the test runner's volatile memory (RAM) for the shortest possible duration necessary to perform the operation. Crucially, these sensitive values must **never be written to disk** (even temporarily), persisted in logs (even encrypted logs without strict access controls), or left lingering in memory beyond their immediate utility. After use, they should be explicitly cleared from memory by the application or garbage collector where possible, or overwritten to prevent memory forensics.
- **Strong, Non-Credential-Based Authentication for Access:** The entities that request secrets (e.g., a CI/CD pipeline job, a containerized test runner, a specific test pod in

Kubernetes, or a cloud virtual machine running tests) must first authenticate securely with the secrets manager itself. This authentication process should ideally be based on robust, intrinsic, and non-credential-based mechanisms tied to the identity of the requesting entity. Examples include:

  - **Cloud IAM Roles/Managed Identities:** Leveraging the inherent identity of a cloud resource (e.g., an AWS EC2 instance profile, Azure Managed Identity) to grant it access based on trusted service principals.
  - **Kubernetes Service Accounts:** Authenticating based on the cryptographic identity of a Kubernetes pod's service account token, which is automatically managed by Kubernetes.
  - **Application-Specific Authentication (e.g., HashiCorp Vault AppRole):** A secure pull-based mechanism where an application receives a unique "role ID" and generates a "secret ID" to authenticate, often in combination with environmental factors. This approach eliminates the need to store static credentials for the *secrets manager itself*, preventing a recursive credential management problem and simplifying the bootstrapping process for test environments.

- **Least Privilege Access Control (Fine-Grained Permissions):** Granular access policies must be applied meticulously within the secrets manager. This ensures that a specific test script or test suite can only access the precise secrets it requires, and absolutely no more. For instance, a test suite for API A should have read-only access to its specific API key, but no access to database credentials or API keys for API B. These permissions should be defined at the path or resource level within the secrets manager, specifying allowed operations (read, list, create, delete, update) for each secret or group of secrets.
- **Encryption at Rest and In Transit:** All secrets must be encrypted at all times throughout their lifecycle. This applies both to secrets **at rest** within the secrets manager's storage backend (using strong cryptographic algorithms, typically managed through a dedicated Key Management Service like AWS KMS or Azure Key Vault's own key management infrastructure) and to secrets **in transit** over the network (using industry-standard, robust protocols like TLS/SSL for all communication between the test runner and the secrets manager's API endpoint). This protects against eavesdropping and unauthorized access to stored data.

- **Comprehensive Audit Logging and Monitoring:** Every single operation performed on secrets within the secrets manager—including secret creation, updates, deletions, and critically, every access attempt (successful or failed)—must be meticulously logged. These logs must be immutable, tamper-evident, and contain rich contextual information (timestamp, source IP, authenticated identity, operation type, affected secret, request ID). These audit trails are fundamental for security monitoring, detecting anomalous behavior, facilitating rapid incident response in the event of a breach, demonstrating compliance with regulatory requirements, and providing a verifiable record for post-incident forensic analysis.
- **Automated Rotation and Lifecycle Management:** Implement robust automated processes for regularly changing credentials, even for static secrets that cannot be dynamically generated. This includes setting scheduled rotations for database passwords, third-party API keys, and other long-lived credentials. Automated rotation significantly reduces the impact of a compromised secret by limiting its active lifespan and continuously keeps credentials fresh, thwarting brute-force attempts or the long-term exploitation of discovered secrets.
- **Secrets Masking in Logs and Outputs:** Under no circumstances should secrets ever be visible in plaintext within build logs, test reports, debugging output, error messages, system logs, or any other persistent output stream. CI/CD systems typically offer built-in features to mask sensitive environment variables or values that match known secret patterns. However, test frameworks themselves must also be designed to explicitly sanitize any output that might inadvertently reveal secret values, ensuring that sensitive data is suppressed or replaced with masked characters (e.g., asterisks).

4. Enterprise-Grade Secrets Management Solutions

The market offers robust, dedicated solutions specifically tailored for secure secrets management, moving far beyond rudimentary environment variables or encrypted files. This section details three prominent and widely adopted options: **HashiCorp Vault**, **AWS Secrets Manager**, and **Azure Key Vault**. The strategic choice among these often hinges on an organization's existing infrastructure, its primary cloud adoption strategy, specific regulatory compliance needs, and architectural preferences for self-managed versus fully-managed services.

4.1. HashiCorp Vault: The Universal Secrets Engine

Overview: HashiCorp Vault is an open-source, API-driven, and highly extensible secrets management tool that stands as a centralized control plane for secrets across diverse environments. It provides a unified interface for storing, generating, and encrypting access to tokens, passwords, certificates, API keys, and other sensitive materials. Vault is designed with a strong focus on security, scalability, and auditability. It offers unparalleled flexibility, capable of being deployed on-premises, in any cloud environment, or consumed as a managed service (e.g., HashiCorp Cloud Platform Vault). Its "universal" nature allows it to integrate with virtually any system that requires secrets.

Key Architectural Features for Testing:

- **Secret Engines (Modular Design):** Vault's core functionality is delivered through modular "secret engines."
    - **KV (Key-Value) Secret Engine (Version 2 Recommended):** This is the most common engine for storing static secrets like third-party API keys, application configurations, or master passwords. KV v2 offers robust features such as **versioning of secrets**, allowing for easy rollback and tracking of changes; **soft-delete capabilities**, providing a recovery window for accidentally deleted secrets; and more robust metadata management, which is crucial for managing evolving test environments and understanding secret provenance.
    - **Dynamic Secret Engines:** A standout feature, these engines enable Vault to generate *on-demand, time-limited credentials* for various backends. This is transformative for testing, as it provides genuinely ephemeral credentials. Examples include:
        - **Database Credentials:** Vault can dynamically create unique database users and passwords (e.g., for PostgreSQL, MySQL, MSSQL, MongoDB) with specific, time-bound access policies. These credentials are valid only for the duration of a test run, significantly reducing the attack surface by ensuring that even if intercepted, they quickly become useless.
        - **Cloud Provider Tokens:** Generation of temporary AWS IAM credentials, Azure service principal tokens, or GCP service account keys.

This allows test environments to securely interact with cloud resources without long-lived access keys.

- **SSH Keys, Certificates:** Issuance of temporary SSH keys or X.509 certificates for secure communication during tests, ideal for ephemeral test infrastructure. This capability fundamentally shifts from storing secrets to *generating* them just-in-time, dramatically minimizing the risk of long-lived, static credential compromise.

- **Authentication Methods (Flexible Identity Verification):** Vault offers a diverse range of authentication methods, making it highly adaptable for CI/CD integration and various test runner environments. This allows Vault to verify the identity of the requesting entity before granting access to secrets.
  - **AppRole:** A secure, pull-based authentication method ideal for machine-to-machine authentication (e.g., CI/CD agents, automated scripts). It involves an administrator configuring a role_id (public identifier) and securely distributing a secret_id (confidential, typically issued once and then renewed or securely transmitted for ephemeral use). The application then combines these to authenticate, avoiding the need to hardcode a Vault token itself.
  - **Kubernetes:** Allows workloads running within Kubernetes clusters to authenticate with Vault based on their Kubernetes service account tokens. This provides a highly secure and automatic authentication mechanism directly from within your containerized test environment, leveraging Kubernetes' built-in identity management.
  - **Cloud IAM/AD (AWS IAM, Azure Active Directory, GCP GCE):** Enables authentication based on the inherent identity of the cloud resource itself (e.g., an AWS EC2 instance profile, an Azure Managed Identity). This is particularly useful for test runners deployed directly on cloud virtual machines or container services, leveraging existing cloud security primitives.
  - **GitHub/LDAP/Userpass:** While less common for automated test pipelines, these methods are valuable for human operators or specific legacy integrations.
- **Policies (Granular Access Control):** Vault's policy system provides fine-grained, path-based access control. Administrators define granular policies that specify precisely what an authenticated entity (whether a human user, an application, or a CI/CD job) can do (read, create, update, delete, list) on specific secret paths or secret engine operations.

This ensures the **Principle of Least Privilege** is strictly enforced, preventing over-permissioning and limiting potential damage in case of a breach.

- **Audit Devices (Immutable Logging):** All requests and responses that pass through the Vault server are meticulously recorded by configurable "audit devices." These logs are designed to be immutable and tamper-evident, providing a complete forensic record of every secret access attempt, operation, and policy evaluation. This rich data can be seamlessly streamed in real-time to **SIEM (Security Information and Event Management) systems** (e.g., Splunk, ELK Stack, Sumo Logic) for centralized security monitoring, real-time alerting on suspicious activities (e.g., excessive secret reads from an unusual IP, failed authentication storms), and long-term storage for compliance reporting.
- **Leasing and Revocation (Secret Lifecycle Management):** Secrets retrieved from Vault, particularly those from dynamic secret engines, are typically associated with a "lease" (a time-to-live, or TTL). Upon lease expiration, the secret is automatically revoked by Vault, rendering it unusable. This inherent lifecycle management feature minimizes the window of exposure if a secret were to be compromised or accidentally left in memory. For static secrets, leases can enforce periodic refreshing, further enhancing security.

Integration Strategy for Test Automation:

1. **Vault Deployment and Configuration:** Organizations first deploy and configure a highly available Vault cluster, often leveraging Kubernetes with the official Helm chart or directly on virtual machines. Key steps involve initializing Vault, securely unsealing it (which can be automated in production environments), and enabling relevant secret engines, such as the KV secret engine for static credentials. Crucially, clear and logical paths for test credentials must be defined (e.g., secret/data/test_suites/api_keys, secret/data/env/staging/db_creds).
2. **Authentication for Test Runners:** The CI/CD agent or test execution environment must be configured to securely authenticate with Vault. For automated pipelines, **AppRole** is a robust choice where the CI/CD system receives a role_id and securely generates or retrieves a secret_id to authenticate. For containerized tests within Kubernetes, leveraging the **Kubernetes authentication method** allows pods to authenticate based on their service account, providing a seamless and secure identity

without any hardcoded credentials. For cloud-native test runners, existing AWS IAM or Azure AD authentication methods would be the preferred secure approach.

3. **Secret Storage:** Sensitive test-specific data, including third-party API keys, database connection strings, credentials for mock services, or test user passwords, are stored securely within Vault paths. Best practice involves organizing these secrets logically by application, environment, or test suite to facilitate granular access control and management.
4. **Test Script Integration:** Test scripts are meticulously designed to programmatically interact with Vault. This involves importing a Vault client library (available for most programming languages, e.g., hvac for Python, vault-java-driver for Java). The script's execution flow begins by authenticating itself with Vault using the configured authentication method. Once successfully authenticated and receiving a Vault token (which is also treated as a secret), the script makes a secure API call to retrieve the necessary credentials from their designated paths. Crucially, secrets are retrieved *just-in-time* for their specific use within a test case or test run, stored only in the test runner's volatile memory, and are explicitly cleared from memory immediately after their function is complete to prevent lingering sensitive data. This includes sanitizing any output or logs to ensure secrets never appear in plaintext.

4.2. AWS Secrets Manager: Native Cloud Credential Management

Overview: AWS Secrets Manager is a fully managed service provided by Amazon Web Services, meticulously designed to help organizations protect access to their applications, services, and IT resources. It offers a comprehensive solution for easily rotating, managing, and retrieving database credentials, API keys, and other secrets throughout their entire lifecycle. Its inherent and deep integration with other core AWS services (such as IAM for access control, KMS for encryption, and CloudTrail for auditing) makes it the most natural and often preferred choice for organizations deeply invested in the AWS cloud ecosystem, providing a seamless and integrated security experience.

Key Features for Testing:

- **Automatic Rotation (Automated Lifecycle):** A highly valuable and differentiating feature, AWS Secrets Manager can automatically rotate secrets for various supported AWS services (e.g., Amazon RDS database credentials, Amazon Redshift credentials, Amazon DocumentDB credentials). This automatic rotation capability ensures that credentials are regularly refreshed without requiring manual intervention or application downtime, which is particularly beneficial for test environments where databases might

be frequently provisioned and de-provisioned, or long-running integration tests are executed against dynamic databases. For custom secrets, it supports rotation via AWS Lambda functions, allowing highly flexible rotation logic.

- **IAM Integration (Fine-Grained Access Control):** Access to secrets stored in Secrets Manager is controlled fundamentally via **AWS Identity and Access Management (IAM)**. This allows for the creation of highly granular permissions based on specific IAM roles, IAM users, and resource-based policies. For test automation, this means your test environment (e.g., an EC2 instance, an ECS task, a Lambda function) can assume an IAM role that has precisely the secretsmanager:GetSecretValue permission only on the specific secrets required for its operations, adhering strictly to the principle of least privilege. This eliminates the need to distribute or embed AWS credentials directly in test code.
- **KMS Encryption (Data Protection):** All secrets stored within AWS Secrets Manager are encrypted both **at rest** using **AWS Key Management Service (KMS)** and **in transit** via TLS/SSL connections. This leverages AWS's robust cryptographic infrastructure, providing strong data protection and meeting many compliance requirements. Organizations can use AWS-managed keys or customer-managed keys (CMKs) in KMS for added control.
- **CloudTrail Logging (Comprehensive Audit):** All API calls made to AWS Secrets Manager, including both management plane operations (e.g., creating a secret, configuring rotation) and data plane operations (e.g., secretsmanager:GetSecretValue – every time a secret is retrieved), are meticulously logged to **AWS CloudTrail**. This provides a complete and unalterable history of who accessed what secret, when the access occurred, and from which source IP address or AWS service/principal. This granular logging is critical for security monitoring, compliance reporting, and forensic analysis, offering deep insight into secret usage.
- **VPC Endpoints (Private Network Access):** AWS Secrets Manager supports **VPC Endpoints (AWS PrivateLink)**, which enables secure and private access to the service directly from within your Amazon Virtual Private Cloud (VPC) without requiring traffic to traverse the public internet. This enhances security by keeping sensitive network traffic within the AWS network and can simplify network configurations for internal test environments that do not require public internet access.

Integration Strategy for Test Automation:

1. **Secret Creation:** Secrets are created within AWS Secrets Manager. Best practice involves storing structured credentials (e.g., an API key and a username, or multiple configuration values) as JSON strings within a single secret. This allows for retrieving multiple related values in one efficient API call, reducing the number of service calls. Secrets are typically named logically, often including environment or application context (e.g., /my_app/test/api_key).
2. **IAM Role for Test Environment:** A dedicated **IAM role** is created for your specific test environment. For example, if tests run on an EC2 instance, an EC2 instance profile would be created; for ECS tasks, an ECS task role; for Lambda functions, a Lambda execution role. This IAM role is then granted secretsmanager:GetSecretValue permission on the specific secret(s) that the test environment is authorized to access, ensuring strict adherence to the principle of least privilege. This is a highly secure approach as the test environment authenticates implicitly via its assigned role, without explicit credentials.
3. **Test Script Integration:** Test scripts running on AWS infrastructure inherently leverage their associated IAM roles or assumed roles to automatically authenticate with AWS services. Test frameworks utilize the **AWS SDK (e.g., Boto3 for Python, AWS SDK for Java, AWS SDK for .NET)** to make secure API calls to Secrets Manager, retrieving secrets by their designated name or ARN. The implementation strategy focuses on retrieving secrets just-in-time for their specific use within a test case or test run, processing them in memory, and immediately clearing the sensitive data once no longer needed or if an error occurs. This prevents lingering secrets in logs, temporary files, or unmanaged memory.

4.3. Azure Key Vault: Microsoft's Cloud Secret Store

Overview: Azure Key Vault is a cloud service provided by Microsoft Azure that offers a secure, centralized store for secrets, cryptographic keys, and SSL/TLS certificates. It is specifically designed to safeguard sensitive information used by cloud applications and services deployed within the Azure ecosystem. Key Vault provides a highly available, scalable, and fully managed solution for managing the lifecycle of cryptographic keys and other secrets, which is crucial for maintaining application security, data privacy, and meeting regulatory compliance requirements in a cloud environment.

Key Features for Testing:

- **Managed Identities for Azure Resources (Azure-Native Authentication):** This feature is a cornerstone of secure Azure application development and is highly beneficial for testing. It enables Azure services (e.g., Azure Virtual Machines hosting test agents, Azure App Services running test harnesses, Azure Functions, Azure Kubernetes Service pods) to authenticate to Key Vault without the need for developers to manually store or manage any credentials in code or configuration files. Azure automatically manages the identity lifecycle (creation, rotation, deletion of the identity's credentials), providing a highly secure and automated way for Azure resources to access secrets. This is the most secure and recommended approach for tests executed within the Azure cloud environment, simplifying identity management significantly.
- **Azure AD Integration (Robust Access Control):** Azure Key Vault integrates directly and seamlessly with **Azure Active Directory (Azure AD)** for strong authentication and granular access control. Access to secrets can be managed via **Azure Role-Based Access Control (RBAC)**, allowing administrators to define precise permissions for specific users, security groups, or service principals at the Key Vault level. Alternatively, traditional Key Vault access policies can be used, granting specific permissions (e.g., "Get," "List" secrets) to defined identities. This comprehensive control ensures that only authorized entities can access sensitive test credentials.
- **Hardware Security Module (HSM) Backing (Enhanced Security):** For organizations with the most stringent security and compliance requirements, Azure Key Vault offers the option to store cryptographic keys in **FIPS 140-2 Level 2 validated Hardware Security Modules (HSMs)**. While more commonly used for master encryption keys, this underlines the robust cryptographic foundation available within Key Vault, offering physical and logical protection for sensitive cryptographic material.
- **Monitoring and Auditing (Comprehensive Logging):** Azure Key Vault integrates directly with **Azure Monitor** for comprehensive logging and monitoring of all Key Vault operations.
    - **Diagnostic Logs:** Key Vault diagnostic logs capture all management plane operations (e.g., creation, deletion, update of secrets) and data plane operations (e.g., retrieval of secrets). These logs include rich contextual information such as the identity of the requester, the operation performed, the resource affected,

and the outcome. These logs can be streamed to **Azure Log Analytics workspaces** for advanced querying and visualization, to Azure Storage Accounts for long-term archival, or directly to **Azure Event Hubs** for real-time integration with external SIEM solutions (e.g., Microsoft Sentinel, Splunk, ArcSight). This integration provides real-time visibility and detailed historical data for auditing and compliance reporting.

- **Soft Delete and Purge Protection (Data Resiliency):** Key Vault includes crucial features like **Soft Delete** and **Purge Protection** that safeguard against accidental or malicious deletion of secrets, keys, or certificates. Soft Delete provides a configurable retention period during which deleted items can be recovered, preventing immediate data loss. Purge Protection, when enabled, further prevents immediate and irreversible deletion, even by privileged users, ensuring data resiliency and aiding in recovery from operational errors or insider threats.

Integration Strategy for Test Automation:

1. **Key Vault Creation:** An Azure Key Vault instance is created within the relevant Azure subscription. This provides the secure, centralized repository for all test-related credentials, such as API keys for test environments, database passwords for staging databases, or credentials for internal testing tools. Secrets are typically named logically to reflect their purpose and scope.
2. **Access Control Configuration:**
    - **Managed Identity:** If your test runner or CI/CD agent is hosted on an Azure resource (e.g., an Azure VM serving as a build agent, a container within an Azure Kubernetes Service cluster), the most secure approach is to enable a **Managed Identity** for that specific resource. This Managed Identity is then granted precise "Get" permissions on the relevant secrets within Key Vault, either through Key Vault access policies or Azure RBAC, strictly adhering to the principle of least privilege.
    - **Service Principal:** For scenarios where Managed Identities are not feasible (e.g., an on-premises CI/CD agent, a custom test environment outside of Azure, or specific cross-cloud integrations), an **Azure AD service principal** is created. This service principal is then granted the necessary permissions to access secrets in Key Vault, and its client ID and client secret (or, for higher security, a certificate) are securely passed to the test runner environment.

3. **Test Script Integration:** Test scripts leverage the **Azure SDK (e.g., azure-keyvault-secrets for secret operations and azure-identity for authentication libraries for Python, Java, .NET, etc.)** to interact with Key Vault. The DefaultAzureCredential class in the azure-identity library is particularly powerful as it automatically attempts various authentication methods (including Managed Identity, environment variables, Azure CLI context, Visual Studio Code context), simplifying the developer experience in Azure-native environments. Once authenticated, secrets are retrieved by their designated name from the Key Vault. Consistent with the other solutions, the emphasis remains on just-in-time retrieval, strictly in-memory handling, and meticulous clearing of sensitive data immediately after its use to prevent any lingering exposure.

5. Practical Implementation Considerations for Test Automation

Integrating a sophisticated secrets manager into a production-grade test automation framework requires careful strategic planning and meticulous implementation beyond the fundamental act of secret retrieval. These considerations are vital to ensure the solution is robust, maintainable, and truly enhances security without inadvertently hindering development velocity or increasing operational complexity.

- **Test Environment Provisioning and Dynamic Secrets:**
    - For sophisticated **integration and end-to-end test environments**, which often require dedicated, isolated infrastructure, organizations should strongly consider adopting highly **dynamic provisioning mechanisms**. When a temporary test environment (e.g., a new database instance, a dedicated API endpoint, a microservice stack) is spun up for a specific test run or a short-lived testing phase, its associated credentials (e.g., a unique database user and password, a temporary API key for a service under test) should ideally be **dynamically generated** at that precise moment.
    - These dynamically generated credentials should then be immediately and securely stored in the secrets manager, often with a very short **lease or time-to-live (TTL)**. Once the test environment is automatically torn down (e.g., by a CI/CD pipeline step), the secrets manager should automatically revoke or expire these temporary credentials, dramatically limiting their window of vulnerability. This pattern is particularly powerful with **HashiCorp Vault's dynamic secrets**

engines, which are purpose-built for this use case, but similar patterns can be architected with cloud-native solutions using Lambda functions or automation. This approach not only enhances security but also ensures test environments are pristine and free from state contamination.

- **CI/CD Pipeline Integration Points:**
  - **Authentication Flow:** The CI/CD agent itself (e.g., Jenkins agent, GitLab Runner, Azure DevOps agent, GitHub Actions runner) is the primary entity that authenticates with the secrets manager. This authentication mechanism (e.g., Vault AppRole, AWS IAM roles, Azure Managed Identity) must be configured meticulously and securely at the pipeline level. It is crucial to ensure that the agent has secure, non-hardcoded access to the secrets manager, often through a bootstrapped identity that is tied to the CI/CD platform's native security features, minimizing the initial attack vector.
  - **Secret Injection and Masking:** Once retrieved by the CI/CD agent from the secrets manager, sensitive credentials should be judiciously **injected into the test runner's environment variables**. Crucially, these environment variables must be rigorously configured within the CI/CD system to be recognized as sensitive and **masked** in all pipeline logs, console outputs, and any generated artifacts. This prevents accidental exposure in build logs, publicly visible run summaries, or debugging outputs. Direct plaintext exposure of secrets in any log or output stream, even transiently, constitutes a critical security failure and must be prevented through automated checks and strict pipeline configuration.
- **Test Data Management vs. Credential Management:**
  - It is vital to distinguish clearly between **sensitive test data** (e.g., Personally Identifiable Information (PII), financial data used for specific test cases, sensitive mock payloads) and **credentials** (authentication materials like API keys, database connection strings, or user passwords for service access). While both categories are sensitive and require protection, a secrets manager is primarily designed for *credentials* and managing their lifecycle.
  - For bulk sensitive test data, the preferred strategy involves **mocking external services**, **anonymization**, or **data obfuscation techniques** to render the data non-sensitive or synthetic for testing purposes. If real sensitive test data is absolutely unavoidable, it should reside in secure, dedicated test data management solutions with stringent access controls and encryption, rather than

overloading a secrets manager which is optimized for small, frequently accessed, and rotating authentication secrets. Using a secrets manager for bulk data is generally an anti-pattern.

- **Performance Considerations and Caching Strategies:**
    - While security is paramount, blindly fetching the exact same secret from the secrets manager for every single test case within a large test suite might introduce unnecessary latency and overhead, particularly in high-volume testing scenarios.
    - A pragmatic balance must be struck: retrieve secrets **once per test run**, **once per logical test group**, or **once per application/test runner startup** within the process. For instance, an API key could be fetched once at the beginning of a test suite and passed to individual tests in memory.
    - Crucially, any caching of secrets must be strictly **in-memory only** and for the shortest possible duration (e.g., the lifespan of a single test run or a specific test class). **Never cache secrets on disk**, in persistent storage, or for indefinite periods, as this immediately reintroduces the very risks (data at rest vulnerabilities) that the secrets manager aims to mitigate. Mechanisms for secure in-memory handling, such as using secure strings or clearing memory regions, should be considered where programming languages support them.
- **Robust Error Handling and Retry Mechanisms:**
    - Test scripts and the underlying test harness must be designed with **graceful error handling** for scenarios where secret retrieval fails. This could be due to transient network issues, temporary unavailability of the secrets manager service, incorrect access permissions configured, or the requested secret not being found.
    - Implement **retry mechanisms with exponential backoff** to handle transient network issues or temporary service interruptions gracefully. For example, if a secrets manager API call fails, the test runner could wait for a short, increasing duration before retrying. However, continuous failures after several retries should lead to a clear failure and alert, as this may indicate a fundamental configuration or access problem. Error messages should be informative but must never leak secret content.

- **Local Development Workflow and Developer Experience:**
    - A secure secrets management strategy must also cater effectively to the **local development workflow** without forcing developers to connect to production-grade secrets managers or requiring complex setups for simple local tests. Overly burdensome security can lead to developers finding insecure workarounds.
    - Provide clear guidance, simplified tools, and secure alternatives for developers to run tests locally:
        - Utilize **local environment variables** for development-specific, non-sensitive credentials or mock values. These should be clearly documented and never committed to version control.
        - Encourage pervasive **mocking or stubbing** of external services during unit tests to entirely avoid the need for real credentials. This also improves test speed and reliability.
        - For integration tests that require real credentials, consider providing access to a **dedicated, isolated development secrets manager instance** (e.g., a developer-friendly Vault instance) or allowing secure loading of development-specific secrets from local, securely managed configuration files (e.g., through tools that inject secrets from a local .env file into a development test run, ensuring these files are excluded from version control).
- **Automated Security Scanning and Linting:**
    - Even with a robust secrets management solution in place, accidents can happen, and new team members might be unaware of best practices. Implement rigorous **automated security scanning tools** (Static Application Security Testing - SAST) and **linting tools** within your CI/CD pipeline.
    - These tools should be specifically configured to proactively scan your entire codebase (including test scripts, build scripts, configuration files, and environment definitions) for any hardcoded secrets, common patterns of sensitive data exposure, or insecure credential practices. This serves as an essential, automated fail-safe mechanism, catching any accidental regressions or security oversights before they are committed to a shared repository, reach production environments, or trigger a costly incident.

- **Principle of Least Privilege (Continuous Review and Enforcement):**
    - The principle of least privilege is not a one-time configuration but an ongoing, dynamic commitment. It requires continuous vigilance. Regularly **review and refine the permissions** granted to your test identities (IAM roles, service principals, Vault AppRoles) within the secrets manager.
    - As your application and test suite evolve, new access might be needed, or old, unnecessary access might become redundant. Conduct periodic security audits, leveraging the secrets manager's audit logs, to verify that only the absolute minimum necessary access is maintained. This continuous scrutiny helps detect and remediate over-permissioned identities, significantly reducing the potential blast radius of a compromised identity and strengthening your overall security posture.

6. Auditability and Compliance: The Unsung Heroes

Beyond preventing direct security breaches, one of the most compelling and often underestimated advantages of using a dedicated secrets manager is the **comprehensive and immutable auditability** it provides. These solutions are built from the ground up with compliance and forensic capabilities in mind, transforming an often opaque area of operations into a transparent and accountable one. Each of the discussed solutions offers robust logging capabilities that are critical for security posture and regulatory adherence:

- **HashiCorp Vault - Detailed Audit Devices:** Vault's architecture includes dedicated "audit devices" that capture every single API request and response that passes through the Vault server. This includes:
    - **Authentication attempts:** Every successful and failed authentication, detailing the method used (e.g., AppRole, Kubernetes), the authenticated identity (e.g., specific role ID, Kubernetes service account name), and the source IP address. This granular detail is crucial for identifying unauthorized access attempts.
    - **Secret operations:** All read, write, update, delete, and list operations on secrets, including the specific secret path accessed, the data being written (often

obfuscated), and the identity performing the action. This creates a detailed log of secret lifecycle management.

- **Policy evaluations:** Logging of how policies were applied to each request, providing transparency into access control decisions. These audit logs are designed to be immutable and tamper-evident, ensuring their integrity for forensic purposes. This rich data can be seamlessly streamed in real-time to **SIEM (Security Information and Event Management) systems** (e.g., Splunk, ELK Stack, Sumo Logic) for centralized security monitoring, real-time alerting on suspicious activities (e.g., excessive secret reads from an unusual IP, failed authentication storms), and long-term storage for compliance reporting and historical analysis.

- **AWS Secrets Manager - Integrated with CloudTrail:** AWS Secrets Manager is inherently integrated with **AWS CloudTrail**, AWS's service for logging API calls and related events across all AWS services.
  - **Management Events:** CloudTrail records all management plane operations specific to Secrets Manager, such as the creation, deletion, modification of secrets, and configuration of secret rotation. This provides oversight of secret lifecycle administration.
  - **Data Events:** Crucially, CloudTrail also records all data plane operations, including every instance of secretsmanager:GetSecretValue (i.e., every time a secret is retrieved). Each log entry contains extensive details: the principal who made the request (e.g., IAM user, IAM role assumed by an EC2 instance), the service accessed, the time of the request, the source IP address, and the specific secret ARN. This provides a complete and unalterable history of who accessed what secret, when the access occurred, and from where, which is critical for establishing a clear chain of custody, performing security investigations, and demonstrating compliance with various industry standards like PCI DSS.
- **Azure Key Vault - Comprehensive Azure Monitor Integration:** Azure Key Vault offers deep and native integration with **Azure Monitor** for comprehensive logging and monitoring of all Key Vault operations.
  - **Diagnostic Logs:** Key Vault diagnostic logs capture all management plane operations (e.g., creating, deleting, updating secrets) and data plane operations (e.g., retrieving secrets, listing secrets). These logs include rich contextual information such as the identity of the requester (e.g., Managed Identity

principal ID, service principal ID), the operation performed, the specific secret affected, and the outcome of the request.

- **Authentication and Authorization:** Details of access attempts, including successful and failed ones, and the specific access policies or RBAC roles that were evaluated for the request. These logs can be efficiently streamed to **Azure Log Analytics workspaces** for advanced querying, custom dashboards, and visualization; to Azure Storage Accounts for cost-effective, long-term archival; or directly to **Azure Event Hubs** for real-time integration with external SIEM solutions (e.g., Microsoft Sentinel, Splunk, ArcSight). This integration provides real-time visibility and comprehensive historical data for auditing and robust compliance reporting.

These detailed **audit trails** are not just a technical feature; they are indispensable operational assets for any organization serious about cybersecurity and regulatory adherence. They provide:

- **Expedited Security Incident Response:** In the unfortunate event of a credential compromise or a suspected breach, the detailed, centralized, and immutable audit logs enable security teams to quickly identify the scope and nature of the compromise, pinpoint the affected secrets, determine the attacker's entry point, and trace their activities within the environment. This significantly accelerates containment efforts, reduces the mean time to recovery (MTTR), and minimizes potential damage.
- **Robust Regulatory Compliance:** Many industry regulations (e.g., PCI DSS, HIPAA, GDPR, ISO 27001, NIST CSF, SOC 2) mandate specific controls around the secure storage, management, and auditing of sensitive information. By providing a clear, verifiable, and tamper-evident record of all secret access and operations, these solutions help organizations demonstrate stringent adherence to these requirements during internal and external audits, thereby avoiding costly financial penalties, severe legal repercussions, and significant reputational damage.
- **Precise Forensic Analysis:** In the immediate aftermath of a security incident, the comprehensive audit logs become the primary source of truth for security analysts. They enable analysts to meticulously reconstruct the sequence of events leading up to the incident, pinpointing vulnerabilities that were exploited, identifying compromised accounts or identities, and understanding the full extent of data exfiltration or system

modification. This detailed forensic capability is vital for post-mortem analysis and strengthening future defenses.

- **Enhanced Operational Visibility and Anomaly Detection:** Continuous monitoring of secret access logs, often integrated with SIEM systems, enables the detection of anomalous or suspicious access patterns. Examples include a sudden spike in secret retrieval attempts from an unusual geographical location, access during non-business hours, retrieval of highly sensitive secrets by an identity that typically doesn't need them, or an unusual number of failed access attempts. Such anomalies can trigger immediate alerts, allowing security teams to proactively investigate and intervene before a full-blown security incident develops, enhancing overall threat detection capabilities.

By diligently leveraging these robust audit capabilities, organizations can transcend basic compliance and move towards a proactive, data-driven approach to security posture management, continuously strengthening their defenses and building a more resilient digital environment.

7. Conclusion: Elevating Test Automation to a Secure Standard

The strategic transition from the perilous practice of hardcoded credentials to the systematic adoption of centralized secrets management in automated test environments represents a pivotal and non-negotiable step in maturing an organization's DevSecOps capabilities. Solutions such as HashiCorp Vault, AWS Secrets Manager, and Azure Key Vault are not merely tools; they are architectural cornerstones that offer robust, scalable, and auditable means of directly addressing the inherent and pervasive security risks associated with sensitive data exposure within testing workflows.

By strategically implementing and diligently utilizing these advanced secrets management tools, organizations can achieve a multitude of critical benefits:

- **Significantly Reduced Attack Surface:** The elimination of plaintext credentials from source code, configuration files, and CI/CD scripts dramatically shrinks the potential points of exploitation for malicious actors. This proactively closes a common and easily exploitable security vulnerability that often remains undetected for prolonged periods.
- **Enhanced Compliance and Regulatory Adherence:** Adopting these solutions directly enables organizations to meet and exceed the stringent requirements of various industry standards and government regulations (e.g., GDPR, HIPAA, PCI DSS, SOC 2). By

providing verifiable proof of secure secret handling, strict access control, and comprehensive audit trails, organizations can navigate audits with greater confidence and mitigate legal and financial risks associated with non-compliance.

- **Streamlined and Automated Credential Lifecycle Management:** The inherent capabilities to automatically rotate, revoke, and manage the complete lifecycle of credentials reduce manual effort, minimize human error, and ensure that secrets are continuously refreshed and secured, even in dynamic test environments. This directly translates to improved operational efficiency, reduced administrative burden, and a significantly lower risk of using stale or compromised credentials.
- **Unprecedented Visibility and Control over Secret Access:** Centralized management coupled with comprehensive audit logging provides an unparalleled level of transparency into who accessed which secret, when, and from where. This real-time visibility is crucial for proactive threat detection, enabling security teams to identify and respond to suspicious access patterns or unauthorized attempts swiftly, before they escalate into full-blown incidents.
- **A Fortified and Trusted Software Delivery Pipeline:** Embedding secure credential handling practices throughout the development and testing phases instills profound confidence in the integrity and resilience of the entire software delivery pipeline. It effectively shifts security from a reactive gate at the end of the process to a proactive, integrated part of every stage, fostering a robust culture of "security by design" that benefits all stakeholders.

Ultimately, secure credential handling in test automation is not just a technical checkbox; it's a strategic imperative that transcends individual security controls. It is about embedding security as a first-class citizen throughout the entire software development lifecycle, fostering a pervasive culture of security responsibility among all stakeholders, and ensuring the continued integrity, reliability, and trustworthiness of software in an increasingly complex and threat-laden digital landscape. The investment in these robust solutions undeniably pays substantial dividends in terms of organizational resilience, compliance assurance, operational efficiency, and ultimately, peace of mind for both development teams and their end-users.

References:

[1] **HashiCorp.** (2020). *Vault: Identity-Based Security for Secrets and Applications.* Retrieved from: https://www.hashicorp.com/resources/vault-identity-based-security

[2] **Amazon Web Services.** (2021). *AWS Secrets Manager Best Practices.* Retrieved from: https://docs.aws.amazon.com/secretsmanager/latest/userguide/best-practices.html

[3] **Microsoft Azure.** (2022). *Azure Key Vault Security Overview.* Retrieved from: https://learn.microsoft.com/en-us/azure/key-vault/general/security-overview

[4] **Rahman, M. M., & Williams, L.** (2019). *Software Developers' Perceptions of Security Tools in Continuous Deployment Pipelines.* In: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE). DOI: 10.1109/ICSE.2019.00250

[5] **Bell, T., & Vemuri, R.** (2018). *Secrets Management: Avoiding Credential Leakage in DevOps Pipelines.* In: Proceedings of the 2018 ACM Workshop on Security in DevOps. DOI: 10.1145/3264888.3264891

[6] **OWASP Foundation.** (2021). *Credential Management Cheat Sheet.* Retrieved from: https://cheatsheetseries.owasp.org/cheatsheets/Credential_Storage_Cheat_Sheet.html

[7] **Paredes, A., & Singh, R.** (2023). *A DevSecOps Approach to Secrets Management in Cloud-Native CI/CD Workflows.* In: Journal of Cloud Computing Advances, 12(1), 34–47. DOI: 10.1186/s13677-023-00301-2

[8] **Palit, P., & Sillito, J.** (2021). *How Developers Manage Secrets in Code: An Empirical Study.* In: Empirical Software Engineering Journal, 26, Article 124. DOI: 10.1007/s10664-021-09980-9

[9] **RedHat.** (2019). *Handling Secrets in Kubernetes Applications.* Retrieved from: https://www.redhat.com/en/blog/how-properly-handle-secrets-kubernetes

[10] **Andreas, J., & George, R.** (2022). *Implementing Secret Rotation in Cloud-Native Pipelines: Practical Lessons from Industry.* In: 2022 IEEE International Conference on Cloud Engineering (IC2E). DOI: 10.1109/IC2E55704.2022.00029

[11] **NIST.** (2020). *Security Considerations for Code Repositories.* NIST Special Publication 800-218. Retrieved from: https://csrc.nist.gov/publications/detail/sp/800-218/final

[12] **Bhat, N., & Kumar, M.** (2017). *Dynamic Secrets for Scalable and Secure Microservices in Test Automation.* In: Proceedings of the 2017 IEEE Symposium on Service-Oriented System Engineering (SOSE). DOI: 10.1109/SOSE.2017.24

Citation: Pradeepkumar Palanisamy. (2023). Secure Credential Handling for Test Automation: A Deep Dive into Vault and Cloud-Native Secrets Managers. International Journal of Computer Engineering and Technology (IJCET), 14(1), 121-147.

Abstract Link: https://iaeme.com/Home/article_id/IJCET_14_01_013

Article Link: https://iaeme.com/MasterAdmin/Journal_uploads/IJCET/VOLUME_14_ISSUE_1/IJCET_14_01_013.pdf

✉ **editor@iaeme.com**