Worse threat

Unauthorized administrative access

General threat description Unauthorized access to the server management interface. Threat agents/Attack vectors Attackers may exploit weak or default credentials, unpatched vulnerabilities, or use social engineering tactics to gain access. Impacts Unauthorized configuration changes, data breaches, and service disruption could occur. Example Attack Scenarios An attacker uses a default credential list to log into a server management console, granting them the ability to change critical settings, exfiltrate data, or cause service outages.

DNS Traffic Encryption

DNS spoofing

General threat description: DNS spoofing, also known as DNS cache poisoning, involves altering DNS records to redirect users from legitimate websites to malicious ones without their knowledge. This can lead to phishing attacks, data theft, and further exploitation.

Threat agents/Attack vectors: Threat agents include cybercriminals and nation-state actors. Common attack vectors involve manipulating DNS cache records on servers or exploiting vulnerabilities in DNS software.

Impacts: The primary impacts are data theft, loss of user trust, and potential financial loss due to phishing or fraudulent activities.

Example Attack Scenarios:

An attacker poisons the DNS cache of a popular banking website, redirecting users to a fake site that captures their login credentials.

A corporate DNS server is compromised, redirecting internal users to a malicious site that installs malware on their systems.

Exploitation of insufficient logging and monitoring

General threat description The absence of adequate logging and monitoring allows attackers to execute malicious actions without detection. This can lead to unauthorized access and potential data breaches with the exploitation remaining hidden from defenders. Threat agents/Attack vectors Cybercriminals may exploit this weakness by injecting malicious code into the API gateway or by initiating a series of unauthorized requests. Without proper monitoring, these activities can go unnoticed. Impacts This threat can lead to compromised systems, unauthorized data access, and potential data loss or manipulation. Moreover, it may result in delayed response to breaches and increased recovery costs. Example Attack Scenarios An attacker successfully bypasses authentication controls and gains access to sensitive data through the API gateway. Due to the absence of logging, the attacker maintains access over an extended period, exfiltrating data without

triggering alerts.

Minimize access to secrets

CIS Benchmark Recommendation id: 5.1.2

Profile Applicability: Level 1 - Master Node

Description: The Kubernetes API stores secrets, which may be service account tokens for the Kubernetes API or credentials used by workloads in the cluster. Access to these secrets should be restricted to the smallest possible group of users to reduce the risk of privilege escalation.

Rationale: Inappropriate access to secrets stored within the Kubernetes cluster can allow for an attacker to gain additional access to the Kubernetes cluster or external resources whose credentials are stored as secrets.

Impact: Care should be taken not to remove access to secrets to system components which require this for their operation.

Audit: Review the users who have get, list or watch access to secrets objects in the Kubernetes API.

Remediation: Where possible, remove get, list and watch access to secret objects in the cluster.

Default Value: By default in a kubeadm cluster the following list of principals have get privileges on secret objects

CLUSTERROLEBINDING SUBJECT TYPE SA-NAMESPACE

cluster-admin system:masters Group

system:controller:clusterrole-aggregation-controller clusterrole-aggregation-controller ServiceAccount kube-system

system:controller:expand-controller expand-controller ServiceAccount kube-system

system:controller:generic-garbage-collector generic-garbage-collector ServiceAccount kube-system

system:controller:namespace-controller namespace-controller ServiceAccount kube-system

system:controller:persistent-volume-binder persistent-volume-binder ServiceAccount kube-system

system:kube-controller-manager system:kube-controller-manager User

References: N/A

Ensure that the Kubernetes PKI key file permissions are set to 600

CIS Benchmark Recommendation id: 1.1.21 Profile Applicability: Level 1 - Master Node

Description: Ensure that Kubernetes PKI key files have permissions of 600. Rationale: Kubernetes makes use of a number of key files as part of the operation of its components. The permissions on these files should be set to 600 to protect their integrity and confidentiality. Impact: None Audit: Run the below command (based on the file location on your system) on the Control Plane node. For example, stat -c '%a' /etc/kubernetes/pki/*.key Verify that the permissions are 600 or more restrictive. or ls -l /etc/kubernetes/pki/.key Verify -rw----- Remediation: Run the below command (based on the file location on your system) on the Control Plane node. For example, chmod -R 600 /etc/kubernetes/pki/.key Default Value: By default, the keys used by Kubernetes are set to have permissions of 600 References:

https://kubernetes.io/docs/admin/kube-apiserver/

Implement rate limiting and resource throttling

Implement and regularly update rate limiting and resource throttling for Redis Server to protect against denial-of-service attacks and resource exhaustion. This control ensures that incoming requests are controlled and that system resource usage (e.g., CPU, memory, I/O) is constrained, preventing overload caused by excessive or malicious traffic. Developers and DevOps engineers should integrate rate limiting policies and resource throttling mechanisms into their Redis deployment using built-in configurations or external middleware, and continuously monitor system performance to adjust thresholds as needed.

Implementation Steps:

Define Rate Limiting Policies: Establish acceptable thresholds for incoming requests based on system capacity and expected workloads.

Implement Rate Limiting Mechanisms: Utilize built-in Redis configurations or deploy external proxies/load balancers that enforce rate limiting on API calls and client connections.

Apply Resource Throttling Controls: Configure settings to limit resource consumption (e.g., CPU, memory, and I/O usage) per client or connection to prevent a single source from overloading the system.

Monitor and Audit: Continuously monitor performance metrics and log access patterns. Regularly review and adjust rate limiting and throttling settings to ensure they remain effective against emerging traffic patterns or attack vectors.

References:

OWASP API Security Project

Enforce secure file permissions on PostgreSQL database files

Implement and regularly update strict file system permissions for PostgreSQL database files to ensure that only authorized users and processes can access or modify them. This control minimizes the risk of unauthorized data tampering and exposure by using OS-level security settings (such as

chmod/chown on Linux) to restrict access to sensitive files. Developers and DevOps engineers should integrate these practices into their deployment procedures, using centralized configuration management tools to enforce and monitor secure file permissions across all database servers.

References:

OWASP Secure Coding Practices - Access Control

Secure authentication using JSON web tokens (JWT) on ReactJS

Implement secure authentication in your React.js application using JSON Web Tokens (JWT) to verify users and maintain session integrity. JWT is a compact, URL-safe method for representing claims to be transferred between two parties. To securely use JWT in React, ensure that the token is stored safely (e.g., in HTTP-only cookies or secure localStorage) and transmitted only over HTTPS. The token should be signed and include an expiration time to prevent unauthorized access in case the token is compromised. Use JWT for stateless authentication in React apps to reduce server-side session management complexity and enhance scalability.

References

Using JWT Authentication in React

Implement role-based access control (RBAC)

To enforce the principle of least privilege and ensure users can only access resources appropriate to their role, implement Role-Based Access Control (RBAC) in your RESTful Web Service. RBAC assigns permissions to roles rather than individual users, making it easier to manage user access at scale. Each user is assigned one or more roles, and those roles determine what actions the user is allowed to perform within the service.

Implementation Steps:

Define Roles: Identify and define roles within your system (e.g., Admin, User, Moderator) based on your service's requirements and user needs.

Assign Permissions to Roles: Map specific actions or resources (e.g., read, write, delete) to each role to control what users can access and modify.

Assign Roles to Users: Assign users to roles based on their job responsibilities and the level of access they require.

Enforce RBAC at the API Level: Ensure that each API endpoint checks the user's roles and verifies whether they have the necessary permissions to perform the requested action.

Monitor and Review Role Assignments: Regularly review and update user roles and permissions to ensure that they align with organizational changes and security best practices.

References: