AWS ECS Fargate Security Configuration Baseline (v2026)

Publication Date: 2026-01-14 **Version:** 2026.1

1. Executive Summary

This document establishes the AWS ECS Fargate Security Configuration Baseline (v2026), a comprehensive set of mandatory controls, architectural patterns, and implementation guidelines designed to ensure the secure deployment and operation of containerized applications on AWS's serverless compute engine [1, 8]. As organizations increasingly adopt Fargate to accelerate development and reduce operational overhead, a standardized and robust security posture is paramount to protect against a sophisticated and evolving threat landscape. This baseline serves as an authoritative resource for engineering teams, cloud architects, and security professionals, providing actionable guidance to build resilient and secure Fargate environments.

The report begins by conducting an advanced threat model of the Fargate ecosystem, analyzing threats through the STRIDE framework and examining specific vectors such as supply chain attacks via malicious base images, secrets exfiltration through insecure handling, runtime exploits leading to lateral movement, and identity-based threats targeting misconfigured IAM roles [28, 29, 31, 33, 39, 48]. The analysis confirms that while Fargate's serverless architecture inherently mitigates entire classes of threats related to host management, significant customer responsibility remains in securing the application, data, identity, and network layers [8].

To address these threats, this baseline prescribes specific, use-case-driven security architectures for three common deployment patterns: internet-facing web applications, internet-facing non-web applications, and internal zero-trust applications. These blueprints leverage a defense-in-depth strategy, integrating services like AWS WAF, Network Load Balancers, VPC Endpoints, and service meshes to enforce strict network segmentation, traffic filtering, and authenticated communication [11, 12].

The core of this document is a set of mandatory security controls organized across six critical domains: Identity and Access Management, Networking, Compute, Storage and Encryption, Observability and Monitoring, and Image Security. Key controls include the rigorous enforcement of least-privilege IAM roles, the isolation of tasks in private subnets, the use of VPC Endpoints for private service communication, the prohibition of privileged containers, the encryption of all data at rest using customer-managed keys, the enablement of runtime threat detection with AWS GuardDuty, and the implementation of a secure software supply chain through image hardening, scanning, and signing.

To facilitate adoption, this baseline provides a detailed implementation guide with both Terraform (Infrastructure as Code) and AWS Management Console examples for each control. A comprehensive audit checklist is also included to enable teams to verify compliance and measure their security posture against these standards. By adhering to the principles and controls outlined in this document, organizations can significantly reduce their attack surface, limit the blast radius of potential security incidents, and confidently leverage the power of AWS Fargate for their mission-critical applications.

2. Introduction

The widespread adoption of containerization has revolutionized how modern applications are developed, deployed, and scaled. AWS Elastic Container Service (ECS) with the Fargate launch type stands at the forefront of this transformation, offering a serverless compute engine that abstracts away the complexities of managing underlying server infrastructure [1, 4]. This allows organizations to focus on application logic and innovation rather than the operational burden of patching, scaling, and securing virtual machines. However, this abstraction does not eliminate security responsibilities; it redefines them. Securing workloads on Fargate requires a deliberate and comprehensive approach that addresses the unique security challenges of a serverless container environment.

The purpose of this AWS ECS Fargate Security Configuration Baseline (v2026) is to provide a production-ready, authoritative standard for securing containerized applications on the Fargate platform. This document consolidates industry best practices, threat intelligence, and architectural patterns into a single, actionable framework. It is intended for a technical audience, including cloud engineers, security architects, DevOps practitioners, and engineering management, who are responsible for the design, implementation, and governance of Fargate-based systems. The scope of this baseline covers the entire lifecycle of a Fargate application, from the construction of secure container images to the configuration of network infrastructure, the enforcement of identity and access controls, and the implementation of runtime monitoring and threat detection.

This document is built upon the foundational principles of the AWS Shared Responsibility Model as it applies to Fargate [6, 8]. While AWS is responsible for the security *of* the cloud—securing the global infrastructure, the Fargate control plane, and the underlying host environment—the customer remains unequivocally responsible for security *in* the cloud [9, 10]. This includes securing application code, managing data encryption, configuring network controls like security groups, and, most critically, defining and enforcing least-privilege Identity and Access Management (IAM) policies [6, 8]. This baseline provides the mandatory controls necessary to fulfill these customer responsibilities effectively.

The report is structured to guide the reader from foundational concepts to practical implementation. It begins with an advanced threat model specific to Fargate, providing the context for the security controls that follow. It then presents secure architectural blueprints for various application use cases, offering tangible design patterns. The core of the document details the mandatory security controls, mapping each to established industry frameworks like the Center for Internet Security (CIS) AWS Foundations Benchmark and NIST SP 800-190 [18, 21]. To ensure these controls are not merely theoretical, a comprehensive implementation guide provides step-by-step instructions using both Terraform and the AWS Management Console. Finally, an audit checklist offers a clear mechanism for verifying compliance. By adopting this baseline, organizations can establish a strong, consistent, and defensible security posture for their Fargate workloads.

3. Advanced Threat Modeling

A systematic approach to threat modeling is essential for understanding the risks inherent in any technology platform and for developing effective mitigation strategies. For AWS Fargate, this involves analyzing potential vulnerabilities across the entire application lifecycle, from the software supply chain to runtime execution. By applying established frameworks like STRIDE and examining common attack vectors, we can identify and categorize the threats that this security baseline is designed to counter.

STRIDE Threat Analysis

The STRIDE framework, which categorizes threats into Spoofing, Tampering, Repudiation, Information Disclosure, Denial of Service, and Elevation of Privilege, provides a structured method for analyzing the security of a Fargate environment [28, 29].

Spoofing threats in a container ecosystem often manifest as the impersonation of legitimate software. Attackers can publish malicious container images to public registries with names that closely mimic official images, a technique known as typosquatting [28]. Unsuspecting developers may inadvertently pull and deploy these compromised images, introducing backdoors or malware into their environment. The primary mitigation against this threat is to establish a trusted software supply chain. This involves whitelisting approved base images from verified publishers, using a private, curated container registry, and leveraging technologies like Docker Content Trust, which uses digital signatures to cryptographically verify the authenticity and integrity of an image's origin [28].

Tampering involves the unauthorized modification of data or code. An attacker could compromise a container registry and alter an existing image, injecting malicious code [28]. They could also manipulate image metadata to trick a system into pulling a compromised version. The defense against tampering is rooted in ensuring immutability and authenticity. Using specific image digests (e.g., `sha256:...`) in deployment configurations instead of mutable tags like `:latest` guarantees that the exact same image is used every time. Docker Content Trust further ensures that the image has not been altered since it was signed by the publisher [28]. Within the build process, avoiding insecure Dockerfile directives like `ADD` for remote URLs can prevent the automatic inclusion of potentially malicious content [28].

Repudiation threats occur when an actor can perform malicious actions and then deny responsibility due to a lack of adequate auditing [28, 29]. To counter this, comprehensive and immutable logging must be implemented across the entire Fargate environment. This includes enabling detailed logging on the container registry (Amazon ECR) to track all image push and pull events, associating them with specific IAM principals. At the AWS account level, AWS CloudTrail must be enabled to capture a complete audit trail of all API calls made to the ECS and other AWS services. Application logs, streamed to Amazon CloudWatch Logs, provide a record of activities within the container itself [28, 29]. By securely collecting and storing these logs, an organization can create an undeniable record of all actions performed.

Information Disclosure is the unauthorized exposure of sensitive data. A common vector for this threat is the insecure handling of secrets, such as API keys or database credentials. These must never be hardcoded into container images or passed as plaintext environment variables, as this makes them easily discoverable [28]. The mandatory mitigation is to use a dedicated secrets management service like AWS Secrets Manager. Secrets are stored securely and injected into the container at runtime, either as files in an in-memory volume or retrieved dynamically by the application. Furthermore, CI/CD pipelines must include automated scanners to detect sensitive data accidentally committed to source code or embedded in images [28].

Denial of Service (DoS) attacks aim to render an application unavailable. In a Fargate context, this can occur if a container consumes excessive resources, crashing itself or impacting the underlying infrastructure, or if a vulnerability in an application dependency is exploited [28, 29]. Fargate provides a strong mitigation at the infrastructure level by requiring users to define strict CPU and memory limits for each task. This isolates the impact of a resource-exhausting container to its own task, preventing it from affecting others. From the customer's side, it is critical to continuously scan all container images for known vulnerabilities (CVEs) in third-party libraries that could be exploited to cause a DoS. Implementing a `HEALTHCHECK` instruction in the Dockerfile also allows the ECS orchestrator to automatically detect and restart unresponsive containers [28, 29].

Elevation of Privilege (EoP) occurs when an attacker gains permissions beyond their authorized level. The most severe form in a container environment is a "container escape," where an attacker breaks out of the container's isolation to gain root access on the host [28, 29]. Fargate's architecture, which runs each task in its own dedicated micro-virtual machine with a separate kernel, provides an exceptionally strong defense against this threat, making traditional container escapes nearly impossible [8]. However, privilege escalation can still occur through the compromise of an overly permissive IAM Task Role. If an attacker gains code execution inside a container, they can steal the credentials associated with the task role and use them to access other AWS services. Therefore, the most critical EoP mitigation in Fargate is the rigorous application of the principle of least privilege to all IAM roles [28, 29]. Additionally, containers should never be run as the root user, a practice enforced by using the `USER` directive in the Dockerfile.

Supply Chain Attacks via Malicious Base Images

The container software supply chain is a significant and attractive target for attackers. A primary vector for compromising this chain is the distribution of malicious base images through public container registries like Docker Hub [31, 33, 34]. Attackers publish images that are disguised as popular, legitimate software but contain hidden payloads such as cryptocurrency miners, backdoors, or credential stealers [31, 33, 34]. Research has shown that such images can be downloaded hundreds of thousands or even millions of times, leading to widespread infections [31, 34]. Attackers often use typosquatting or impersonation to trick developers into using their compromised images [31].

Mitigating this threat requires a defense-in-depth approach focused on establishing a secure and trusted supply chain. Organizations must shift their mindset to treat all public registries as inherently untrusted. The foundational control is to establish a private, internal container registry (such as Amazon ECR) that serves as the single source of truth for all images deployed in production. No image should be admitted to this trusted registry without undergoing a rigorous vetting process. This process must include comprehensive vulnerability scanning using tools that can perform both static analysis (to find known CVEs and misconfigurations) and dynamic analysis in a sandboxed environment (to detect malicious runtime behavior) [31, 35].

Further strengthening the supply chain involves cryptographic verification of image integrity and provenance.

Technologies like Docker Content Trust or Sigstore allow organizations to digitally sign their container images [31, 34]. The deployment pipeline must then be configured to verify these signatures, ensuring that only images from a trusted publisher that have not been tampered with are allowed to run. Development practices must also be hardened. Using minimal base images, such as "distroless" or Alpine, drastically reduces the attack surface by eliminating unnecessary tools and libraries [32, 33, 34]. Pinning base images to a specific, immutable digest (`sha256:...`) rather than a mutable tag like `:latest` ensures build reproducibility and prevents the silent introduction of a compromised image layer.

Secrets Exfiltration

The management of sensitive data, such as database credentials, API keys, and TLS certificates, is a critical security challenge in any application architecture. In containerized environments like Fargate, a common but dangerous practice is to inject these secrets into containers as environment variables [39, 40]. While convenient, this method creates significant risks and multiple vectors for secrets exfiltration. The primary risk is that environment variables are highly visible. Any user or process with permission to inspect the task definition via the AWS API or CLI can view the plaintext values of these secrets [37, 39]. An attacker who compromises a developer's workstation or gains read-only access to the ECS API could easily harvest these credentials.

Furthermore, environment variables are frequently captured in application logs, particularly during startup or in crash dump reports. If logging is not carefully configured, sensitive credentials can be inadvertently written to log files, which may be stored in less secure locations or be accessible to a wider audience than intended [39, 40]. Within the container itself, any process can typically read its own environment variables, making them readily available to an attacker who achieves remote code execution [39, 40].

To mitigate these risks, the mandatory best practice is to completely avoid injecting sensitive data as environment variables. A more secure method is to mount secrets as files into an in-memory volume (like `tmpfs`) within the container [37, 39]. The application is then configured to read the credentials from these files. This prevents the secret values from being exposed in the task definition API response or in the process environment. For the highest level of security, organizations must use a dedicated external secrets management system like AWS Secrets Manager or HashiCorp Vault [40]. These services provide centralized management, fine-grained access control via IAM, automatic secret rotation, and detailed audit trails. The application, using its IAM Task Role, can then dynamically fetch the secrets on-demand at runtime, ensuring they are only held in memory for the briefest possible time.

Runtime Exploits and Lateral Movement

Runtime exploits, particularly those leading to "container escape," are among the most severe threats to containerized environments [41, 42]. These attacks leverage vulnerabilities in the container runtime, the host kernel, or system misconfigurations to break out of the container's isolated sandbox and gain access to the underlying host [42, 44]. Once an attacker has escaped, they can potentially access other containers, escalate privileges, and begin lateral movement across the cloud environment.

The architecture of AWS Fargate provides powerful, built-in protection against this class of threat. By running each task in its own isolated, hardware-virtualized micro-VM, Fargate ensures that tasks do not share an underlying kernel [8]. This strong isolation boundary effectively mitigates the risk of a kernel exploit in one task affecting any other, a primary concern in traditional, multi-tenant container hosts. However, a compromise within a Fargate task can still serve as a dangerous initial foothold for an attacker.

Once an attacker achieves code execution inside a Fargate container, their primary objective shifts to lateral movement within the AWS account. The most common technique is to steal the temporary credentials associated with the task's IAM role [45]. These credentials can be accessed by querying the EC2 metadata service endpoint, which is available to the container. If the IAM Task Role is overly permissive, the attacker can use these stolen credentials with the AWS CLI or SDKs to interact with other AWS services. They can read data from S3 buckets, access databases in RDS, or launch new EC2 instances [45]. From there, they may attempt to escalate their privileges further by identifying and exploiting misconfigured IAM policies that allow them to assume more powerful roles [45]. Therefore, the most critical defense against lateral movement from a compromised Fargate task is the rigorous application of the principle of least privilege to the IAM Task Role, severely limiting what an attacker can do even if they gain initial access. Runtime threat detection tools, such as AWS GuardDuty, are also essential for identifying anomalous behavior post-compromise, such as unexpected process execution or communication with known command-and-control servers [41, 42].

Identity-Based Threats and IAM Task Roles

In the AWS cloud, identity is the primary security perimeter. For AWS Fargate, the IAM Task Role is the principal identity assigned to a running application, granting it the permissions to interact with other AWS services [47, 49, 50]. Consequently, identity-based threats, particularly those stemming from compromised or overly permissive IAM roles, are a paramount concern. A misconfigured task role can provide an attacker with a direct and powerful pathway to access, modify, or exfiltrate sensitive data and resources across the entire AWS account. Securing the IAM Task Role is therefore one of the most critical security responsibilities for any team deploying workloads on Fargate.

The foundational best practice is to grant only the absolute minimum permissions required for the task to perform its specific, intended function [47, 49, 50]. This means strictly avoiding the use of wildcards (`*`) in IAM policy statements and instead explicitly defining the precise actions (e.g., `s3:GetObject`) and the full Amazon Resource Names (ARNs) of the resources the task needs to access. Tools like IAM Access Analyzer can help generate least-privilege policies by analyzing CloudTrail logs to determine the exact API calls an application makes [47, 50]. This helps to tailor policies accurately and eliminate unused permissions, which represent a potential vector for privilege escalation.

Beyond the permissions policy, the role's trust policy (or assume role policy) must also be secured to prevent the "confused deputy" problem [48]. This is a security vulnerability where a legitimate entity with permission to assume a role is tricked by a malicious actor into misusing its authority. To mitigate this, the trust policy for an ECS task role must include condition keys like `aws:SourceAccount` and `aws:SourceArn` [48]. The `aws:SourceAccount` condition ensures that only ECS tasks from your specific AWS account can assume the role. The `aws:SourceArn` condition provides even more granular control, restricting assumption to tasks belonging to a specific ECS service or cluster within that account. This prevents an unauthorized service, even within the same account, from illegitimately acquiring the role's permissions. Regular auditing of role usage and the consistent management of IAM policies through Infrastructure as Code (IaC) are essential disciplines for maintaining a robust identity security posture.

4. Use-Case Specific Architectures

Securing AWS Fargate workloads is not a one-size-fits-all endeavor. The optimal security architecture depends heavily on the application's use case, particularly how it interacts with users and other services. This section details three distinct, production-ready architectural patterns: one for internet-facing web applications, one for internet-facing non-web applications, and one for internal applications designed with a zero-trust security model. Each pattern employs a defense-in-depth strategy tailored to its specific threat profile.

Architecture for Internet-Facing Web Applications

Figure 1: Internet-Facing Web Application Architecture with ALB, WAF, and Multi-AZ Deployment

The deployment of internet-facing web applications, such as public websites or APIs, on AWS Fargate demands a robust architecture that prioritizes security at every layer. As shown in Figure 1 above, this pattern is designed to securely expose HTTP/S services to the internet while shielding the backend application components from direct attack. The architecture is built within an Amazon Virtual Private Cloud (VPC) and leverages a suite of AWS services to create multiple layers of defense.

The core architectural blueprint designates an Application Load Balancer (ALB) as the sole ingress point for all internet traffic [2, 51]. The ALB is a managed Layer 7 load balancer that is placed in public subnets across multiple Availability Zones for high availability [51]. These public subnets have a route to an Internet Gateway, allowing them to receive traffic from the public internet [51]. The Fargate tasks themselves, which

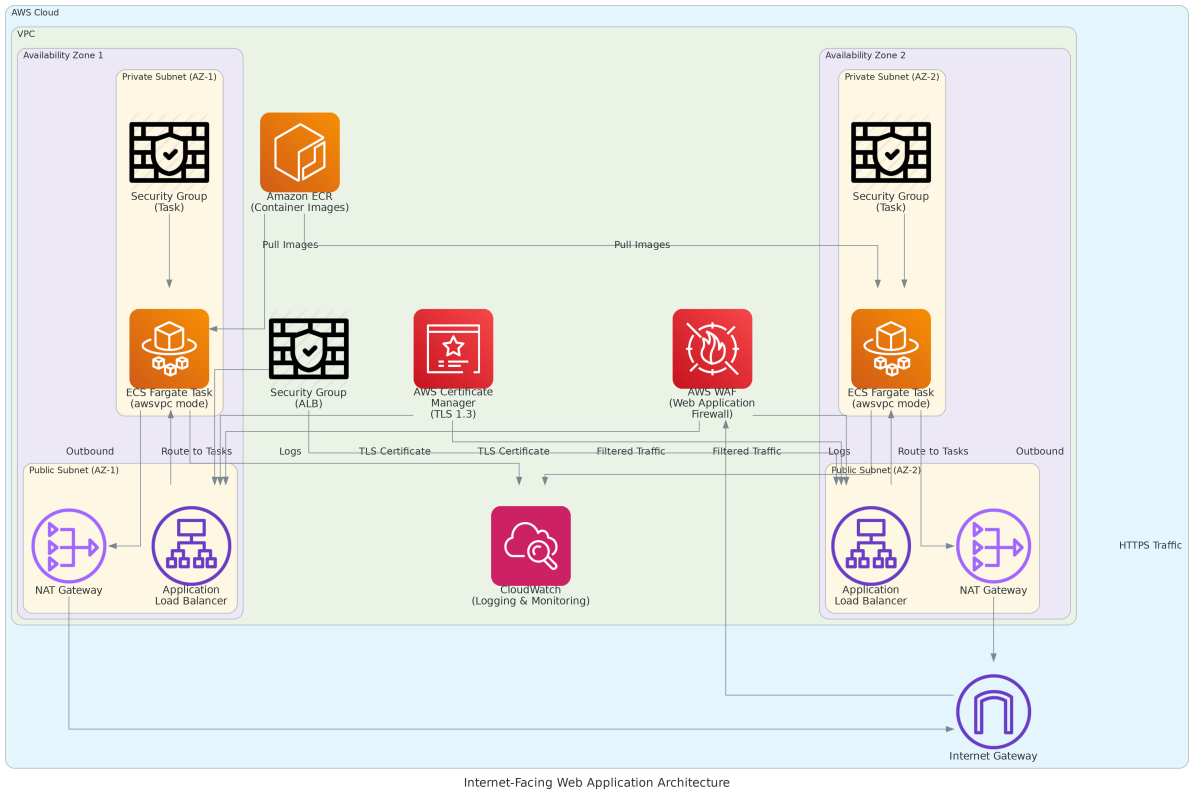

Figure 1: Internet-Facing Web Application Architecture

run the containerized application code, are deployed into private subnets [72]. These subnets have no direct route to the Internet Gateway, making the tasks completely inaccessible from the public internet. This network segmentation is a fundamental security control, ensuring that all inbound traffic must first pass through the ALB. For tasks that require outbound internet connectivity, such as calling a third-party API, a NAT Gateway is placed in a public subnet, and the private subnets' route tables are configured to direct outbound traffic to it [59].

Security is enforced at multiple points. The ALB's security group is configured to allow inbound traffic from anywhere (`0.0.0.0/0`) but only on ports 80 (HTTP) and 443 (HTTPS) [51]. The Fargate tasks' security group is then configured with a highly restrictive rule that allows inbound traffic only from the ALB's security group, effectively creating a private communication channel [51]. To protect data in transit, the ALB's HTTPS listener must be configured to terminate TLS connections using a certificate from AWS Certificate Manager (ACM) [55, 56]. A rule should be added to redirect all HTTP traffic to HTTPS, and a strong TLS security policy, such as `ELBSecurityPolicy-TLS13-1-2-2021-06`, must be enforced to mandate the use of modern, secure ciphers and protocols [56, 65, 69].

For protection against application-layer attacks, AWS Web Application Firewall (WAF) must be integrated with the Application Load Balancer [52, 63]. WAF inspects incoming HTTP/S requests and can block malicious traffic based on a set of rules. It is mandatory to use AWS Managed Rule Groups, such as the `AWSManagedRulesCommonRuleSet` (which protects against OWASP Top 10 threats), `AWSManagedRulesSQLiRuleSet`, and `AWSManagedRulesXSSRuleSet` [60]. These pre-configured rule sets provide immediate, broad-spectrum protection against common exploits. Custom rules, such as rate-based rules to mitigate HTTP floods and IP set match rules to block known malicious actors, should be added to provide tailored protection for the specific application [63].

Architecture for Internet-Facing Non-Web Applications

Figure 2: Internet-Facing Non-Web Application Architecture with NLB and DDoS Protection

Many organizations need to run internet-facing services that do not use standard web protocols like HTTP/S. These non-web applications, which include real-time gaming servers, IoT data ingestion endpoints, or custom TCP/UDP services, require a different architectural pattern for handling ingress traffic. As illustrated in Figure 2, while the core security principles of network isolation and least privilege remain the same, the implementation at the load balancing layer is distinct.

The key component in this architecture is the Network Load Balancer (NLB), which replaces the Application Load Balancer [75]. The NLB is a high-performance Layer 4 load balancer capable of handling TCP and UDP traffic with very low latency [75]. It is placed in the public subnets of the VPC and configured with listeners for the specific ports required by the application. Similar to the web application pattern, the Fargate tasks running the application code are deployed in private subnets to shield them from direct internet access [75]. The NLB forwards traffic directly to the tasks registered in its target group, serving as the single, controlled entry point.

This architectural choice has significant security implications. Because the NLB operates at the transport layer, it performs a direct passthrough of the connection and does not terminate TLS or inspect application-layer content [66]. This means that the responsibility for encryption shifts to the application itself. If the protocol requires encryption, the application code running inside the Fargate container must handle the entire TLS handshake, manage the certificate and private key, and perform the encryption and decryption of traffic [66]. This requires a secure mechanism, such as AWS Secrets Manager, to distribute and rotate the TLS certificates to the tasks [66].

Furthermore, the inability to inspect Layer 7 traffic means that AWS WAF cannot be used with an NLB. The application is therefore not protected by WAF's capabilities for filtering malicious requests like SQL injection. The responsibility for input validation, sanitization, and defense against all application-layer attacks falls entirely on the application code. Developers must implement robust security coding practices to compensate for the lack of a web application firewall. For protection against network-layer DDoS attacks, the architecture benefits from AWS Shield Standard, which is automatically included and provides defense against common

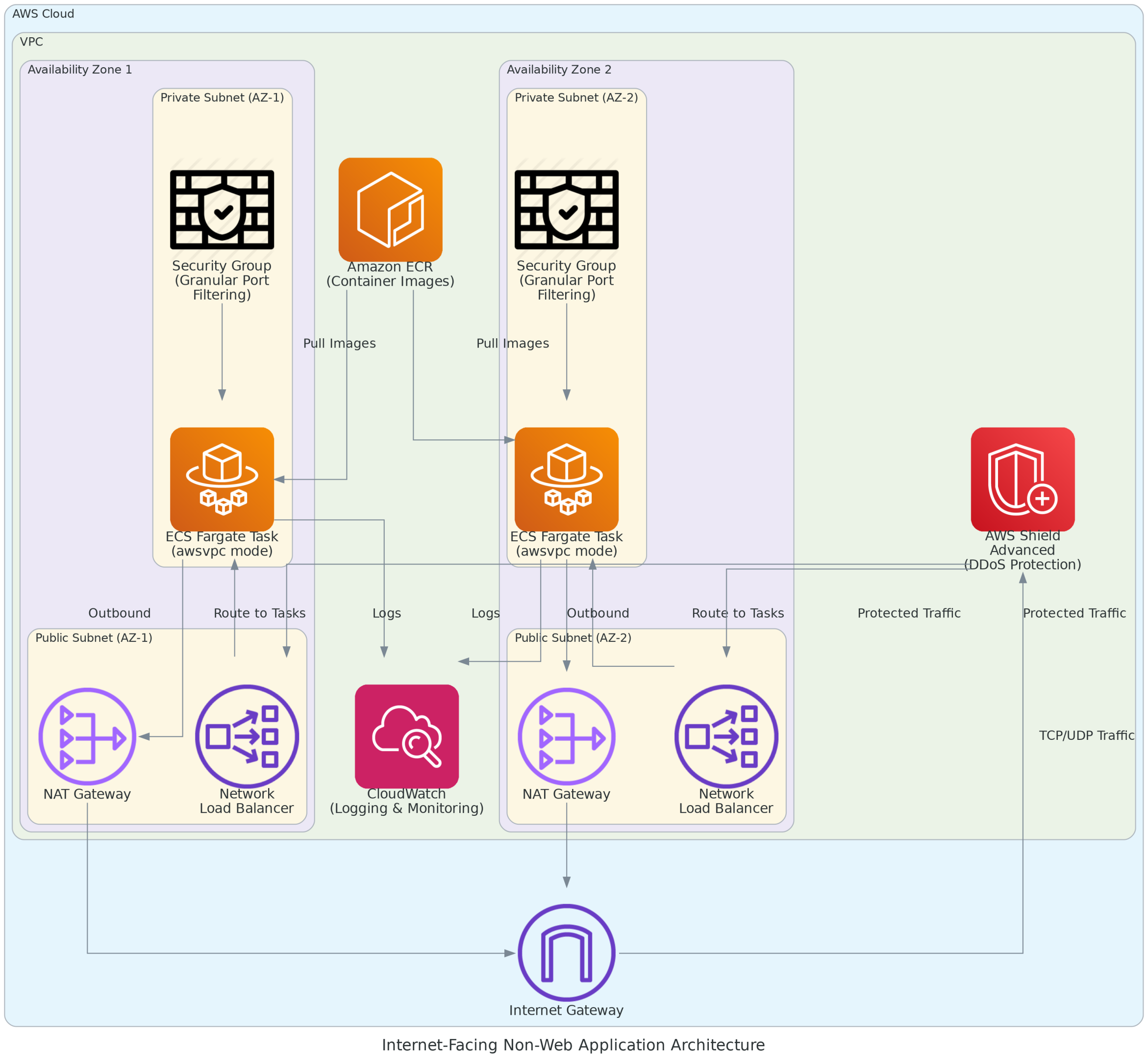

Figure 2: Internet-Facing Non-Web Application Architecture

volumetric attacks. The security group for the Fargate tasks must be configured to allow inbound traffic on the application port(s) only from the source IP addresses of the NLB nodes or, if applicable, from a specific set of trusted client IP ranges [75].

Architecture for Internal Zero-Trust Applications

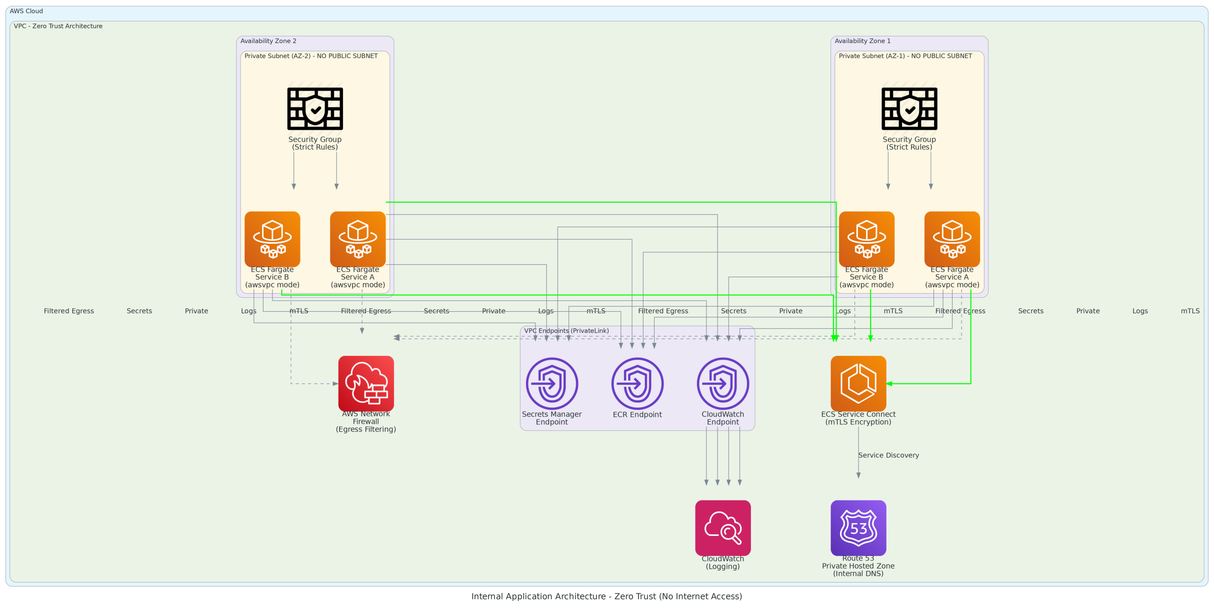
AWS Cloud
VPC - Zero Trust Architecture
Availability Zone 2
Private Subnet (AZ-2) - NO PUBLIC SUBNET
Security Group (Strict Rules)
ECS Fargate Service B (awsvpc mode)
ECS Fargate Service A (awsvpc mode)
Availability Zone 1
Private Subnet (AZ-1) - NO PUBLIC SUBNET
Security Group (Strict Rules)
ECS Fargate Service B (awsvpc mode)
ECS Fargate Service A (awsvpc mode)
Filtered Egress
Secrets
Private
Logs
mTLS
Filtered Egress
Secrets
Private
Logs
mTLS
Filtered Egress
Secrets
Private
Logs
mTLS
Filtered Egress
Secrets
Private
Logs
mTLS
VPC Endpoints (PrivateLink)
AWS Network Firewall (Egress Filtering)
Secrets Manager Endpoint
ECR Endpoint
CloudWatch Endpoint
ECS Service Connect (mTLS Encryption)
Service Discovery
CloudWatch (Logging)
Route 53 Private Hosted Zone (Internal DNS)
Internal Application Architecture - Zero Trust (No Internet Access)

Figure 3: Internal Application Architecture (Zero Trust)

Figure 3: Internal Application Architecture with Zero Trust and AWS PrivateLink

In a modern enterprise, many applications are intended for internal use only and must never be exposed to the public internet. Securing these applications requires a zero-trust architecture, which operates on the principle of "never trust, always verify." This model assumes no user or service is inherently trustworthy, regardless of its network location, and requires strong authentication for every connection [80]. The architecture diagram (Figure 3) illustrates a pattern for deploying internal Fargate applications in a highly isolated, VPC-only environment that embodies zero-trust principles.

The foundation of this architecture is a VPC that is completely sealed off from the public internet. This is achieved by designing a VPC that contains only private subnets and has no Internet Gateway (IGW) or NAT Gateway attached [86]. The absence of these components makes it physically impossible for any resource within the VPC to be reached from the internet or to initiate outbound connections to the internet. This powerful control eliminates the threat of external attacks and prevents data exfiltration to unauthorized endpoints [86]. To allow Fargate tasks to communicate with essential AWS services (like ECR for pulling images or CloudWatch for sending logs), VPC Endpoints powered by AWS PrivateLink are used [77, 86]. These create private, secure connections between the VPC and AWS services without traffic ever traversing the public internet [77]. Interface endpoints for ECR, CloudWatch Logs, and Secrets Manager, along with a gateway endpoint for S3, are mandatory for a Fargate deployment in this isolated environment [74, 77].

For service-to-service communication within this sealed network, a zero-trust model requires strong, authenticated connections. This is best achieved using a service mesh like AWS App Mesh, which enforces mutual TLS (mTLS) for all inter-service communication [81, 89, 92]. App Mesh works by deploying a lightweight Envoy proxy as a sidecar container alongside each application container [89, 92]. This proxy intercepts all network traffic and automatically handles the mTLS handshake, certificate validation, and encryption, without requiring any changes to the application code [81]. By enforcing mTLS, the service mesh ensures that even if an attacker gains a foothold within the network, they cannot spoof service identities or eavesdrop on traffic, severely limiting their ability to move laterally [81].

For internal applications that require access by human users, an Identity-Aware Proxy (IAP) pattern provides another powerful zero-trust control [80]. In this model, an internal Application Load Balancer is placed in front of the Fargate service. The ALB is configured to authenticate users via an identity provider like Amazon Cognito or another OIDC-compliant provider [57]. When a user attempts to access the application, the ALB intercepts the request and redirects them to the identity provider for login. Only after successful authentication does the ALB forward the request to the backend Fargate application, passing along the user's verified identity in a secure HTTP header [57]. This moves authentication to the network edge and ensures that no unauthenticated traffic can reach the internal service, enforcing per-request authorization based on identity rather than network location [80].

5. Mandatory Security Controls

This section details the mandatory security controls required to establish a secure baseline for all AWS ECS Fargate workloads. These controls are organized into six critical domains, reflecting a defense-in-depth strategy that spans the entire application stack. Each control is described in detail, including the specific threats it mitigates and its alignment with established industry security benchmarks. Adherence to these controls is essential for protecting containerized applications from a wide range of threats.

Identity and Access Management

Effective Identity and Access Management (IAM) is the cornerstone of cloud security. For Fargate, this means enforcing the principle of least privilege on all identities associated with a task to minimize the potential impact of a compromise.

The most critical control in this domain is to **Enforce Least Privilege for Task and Task Execution Roles**. Every Fargate task uses two distinct IAM roles: the Task Execution Role, used by the ECS agent to manage the task lifecycle (e.g., pull images, send logs), and the Task Role, used by the application code inside the container to access other AWS services [99, 100]. These roles must be separate and configured with custom, narrowly scoped IAM policies [99, 100]. This control directly mitigates the threats of privilege escalation and lateral movement [100]. If a container is compromised, a tightly scoped Task Role prevents the attacker from accessing unauthorized AWS resources. For example, an attacker inside a web server container should not be able to access a production database if the Task Role does not explicitly grant that permission. Implementation requires creating custom IAM policies that specify the exact actions and the full Amazon Resource Names (ARNs) of the resources each role needs [100]. Wildcards (`*`) must be avoided in production policies. The trust policy for these roles must also be configured to allow only the `ecs-tasks.amazonaws.com` service principal to assume them, with conditions to restrict this to a specific source account and ARN [99].

Networking

Networking controls for Fargate focus on creating isolated environments and filtering traffic to reduce the application's attack surface. By leveraging VPC-native features, we can build a secure network foundation for our containerized workloads [78, 87].

A fundamental control is to **Isolate Tasks in Private Subnets without Public IPs**. All production Fargate tasks must be launched into private subnets, and the option to assign a public IP address must be disabled [88, 102]. This ensures tasks are not directly addressable from the public internet, shielding them from external scans and direct attacks. This control is a primary defense against unauthorized network access, direct vulnerability exploitation, and certain types of Denial-of-Service attacks. For services that need to be exposed to the internet, an Application Load Balancer in public subnets should act as the controlled ingress point. Implementation is achieved in the service's network configuration by selecting only private subnet IDs and setting the `assignPublicIp` parameter to `DISABLED` [103].

To facilitate secure communication with other AWS services from within this isolated environment, the control to **Utilize VPC Endpoints for Private AWS Service Access** is mandatory. Instead of routing traffic to services like ECR, S3, or Secrets Manager through a NAT Gateway and the public internet, VPC Endpoints must be used [88]. This creates a private network path that keeps all traffic within the secure AWS backbone, mitigating the threat of data interception or man-in-the-middle attacks [88]. Endpoint policies can provide an additional layer of security by restricting access to specific resources, such as allowing pulls only from a designated ECR repository. Implementation requires creating interface endpoints for services like ECR and CloudWatch Logs and a gateway endpoint for S3 within the VPC [88].

The final networking control is to **Implement Least-Privilege Security Groups**. Security groups act as stateful firewalls at the task's network interface level and must be configured with the most restrictive rules possible [78]. This control is essential for preventing unauthorized lateral movement within the VPC. If a container is compromised, a restrictive security group will prevent the attacker from scanning for and connecting to other resources on the network, such as databases or other microservices [88]. For a service fronted by an ALB, the task's security group should only allow inbound traffic from the ALB's security group on the specific application port. Outbound rules should be equally restrictive, permitting traffic only to the specific destinations the application needs to reach.

Compute

While Fargate abstracts the underlying host, security controls at the compute layer remain critical for protecting the container execution environment. These controls leverage Fargate's inherent security features and augment them with runtime monitoring.

The first control is to **Prohibit Privileged and Host-Access Containers**. This is a control that Fargate enforces by design [8, 106, 107]. The platform inherently prohibits running containers with the `--privileged` flag, mounting host volumes, or adding most Linux capabilities [8]. This design choice is a powerful mitigation against the threat of container breakout, where an attacker escapes the container's isolation to compromise the underlying host [8]. The implementation of this control is to exclusively use the Fargate launch type for all containerized workloads, thereby inheriting these built-in protections [8].

Next, it is mandatory to **Enforce Task-Level CPU and Memory Resource Limits**. Every Fargate task definition must include specific, mandatory allocations for CPU and memory. These are not just performance settings; they are security controls that mitigate the threat of resource-exhaustion Denial-of-Service attacks [8]. By setting hard limits, the impact of a malfunctioning or compromised container that enters an infinite loop is confined to the individual task, which will be terminated and restarted by ECS, preserving the stability of the overall service. Implementation requires defining appropriate `cpu` and `memory` values in the task definition based on application performance testing.

The third compute control is to **Enable Runtime Threat Detection with GuardDuty**. AWS GuardDuty Runtime Monitoring must be enabled for all Fargate clusters [104]. This service deploys a lightweight, managed security agent as a sidecar container that monitors the task's operating system and container-level activity for malicious behavior [104]. This control provides a critical detection layer for threats that occur post-compromise, such as malware execution, cryptocurrency mining, privilege escalation attempts within the container, and anomalous network connections indicating command-and-control communication [104]. Implementation involves enabling GuardDuty in the AWS account and activating the Runtime Monitoring feature for ECS [104, 105].

Storage and Encryption

Securing data at rest is a critical component of a defense-in-depth strategy. This involves encrypting all storage associated with a Fargate task and securely managing application secrets.

The first control is to **Encrypt Ephemeral Storage with Customer-Managed Keys (CMKs)**. While Fargate encrypts the ephemeral storage attached to each task by default with an AWS-managed key, it is mandatory to use a Customer-Managed Key from AWS KMS for workloads handling sensitive data [108, 111]. This provides a higher level of security, control, and auditability. Using a CMK allows you to manage the key's lifecycle, including rotation and revocation, and provides a detailed audit trail in CloudTrail of every access attempt [108]. This control mitigates the threat of unauthorized data access to the ephemeral storage layer. Implementation requires creating a CMK in KMS and configuring the ECS cluster to use this key for its managed storage configuration [108].

The second, and equally critical, control is to **Utilize External and Dedicated Secret Stores**. Application secrets, such as database credentials and API keys, must never be stored in container images or passed as plaintext environment variables [109, 126]. They must be stored in a dedicated service like AWS Secrets Manager or AWS Systems Manager Parameter Store (SecureString type) [109]. This control directly mitigates the significant threat of secret exposure and exfiltration [109]. Storing secrets externally centralizes their management, enables automatic rotation, and provides a robust audit trail. The application running in the Fargate task must be configured to retrieve these secrets dynamically at runtime using its IAM Task Role, ensuring secrets are only held in memory.

Observability and Monitoring

Comprehensive observability is a critical security function, providing the visibility needed to detect anomalies, investigate incidents, and verify that security controls are functioning as intended.

A mandatory control is to **Enable Comprehensive Logging with Container Insights**. All Fargate services must have AWS CloudWatch Container Insights enabled, and all tasks must be configured to stream their logs to CloudWatch Logs using the `awslogs` driver [112, 113, 114]. This provides a centralized, durable record of all container activity and detailed performance metrics. This control is fundamental for incident detection and response, as it provides the data needed for forensic analysis and can reveal security anomalies like sudden spikes in CPU usage indicative of cryptomining [112]. Implementation requires enabling Container Insights at the cluster level and configuring the `logConfiguration` in the task definition.

To gain visibility into network activity, the control to **Monitor Network Traffic with VPC Flow Logs** must be implemented. VPC Flow Logs must be enabled for the VPCs where Fargate tasks are deployed [112]. This feature captures metadata about all IP traffic to and from the tasks' network interfaces. These logs are essential for detecting network-based threats like internal reconnaissance (port scanning), data exfiltration attempts to unexpected external IPs, and anomalous traffic patterns that could indicate a command-and-control channel [112]. Implementation involves creating a flow log for the VPC and publishing the logs to CloudWatch Logs or an S3 bucket for analysis.

Image Security

The security of a containerized application is fundamentally dependent on the security of its container image. A secure software supply chain for images is non-negotiable.

The first control is to **Harden Base Images and Use Multi-Stage Builds**. All container images must be built upon a minimal, hardened base image (e.g., “distroless” or Alpine) [116]. Dockerfiles must use multi-stage builds to ensure the final runtime image contains only the application and its direct dependencies, excluding all build tools, compilers, and shells. This control mitigates the threat of vulnerability exploitation by drastically reducing the image's attack surface and denies attackers the tools they would use for post-exploitation activities [116].

The second control is to **Scan Images for Vulnerabilities and Enforce Signing**. All container images must be scanned for known vulnerabilities before deployment using a tool like Amazon ECR's Enhanced Scanning [118, 119]. This proactive measure prevents the deployment of applications with known security flaws [118]. Additionally, a mechanism for cryptographically signing images, such as AWS Signer or Sigstore, must be implemented [116, 117]. Image signing mitigates the threat of image tampering by providing cryptographic proof of an image's integrity and provenance [117]. The deployment pipeline must be configured to verify these signatures before allowing a deployment [120].

Finally, for every container image, a **Generate and Maintain a Software Bill of Materials (SBOM)** must be created. An SBOM is a formal, machine-readable inventory of all software components and dependencies within the image [116]. This control mitigates the threat of poor visibility into the software supply chain. When a new zero-day vulnerability is discovered, an SBOM allows an organization to immediately and accurately identify every affected application, drastically reducing the time to remediation [117]. Implementation involves integrating an SBOM generation tool like Syft into the CI/CD pipeline and storing the resulting SBOM artifact alongside the container image [116].

6. Control Matrix Table

The following table summarizes the mandatory security controls for AWS ECS Fargate, detailing the threats they mitigate, their mapping to industry-standard benchmarks, and their implementation priority. Priority is categorized as **High** (foundational controls that must be implemented immediately), **Medium** (controls that add significant security layers), and **Low** (controls that enhance or mature the security posture).

Control Name	Threat(s) Mitigated	CIS AWS Foundations v6.0.0 Mapping [121-124]	NIST SP 800-190 Mapping [125-129]	Priority
Enforce Least Privilege for Task and Task Execution Roles	Privilege Escalation, Lateral Movement, Unauthorized Data Access	1.5, 1.16	3.3.1 (Unbounded Administrative Access)	**High**
Isolate Tasks in Private Subnets without Public IPs	Unauthorized Network Access, Vulnerability Exploitation, DoS	4.1, 4.2 (Principle)	3.3.3 (Poorly Separated Inter-Container Network Traffic)	**High**
Utilize VPC Endpoints for Private AWS Service Access	Data Exfiltration, Man-in-the-Middle Interception	4.5 (Principle)	3.3.3 (Poorly Separated Inter-Container Network Traffic)	**High**
Implement Least-Privilege Security Groups	Unauthorized Lateral Movement, Network Reconnaissance	4.3	3.3.3, 3.4 (Container Security)	**High**
Prohibit Privileged and Host-Access Containers	Container Breakout, Host Compromise	N/A (Platform-enforced)	3.1.2 (Configuration Defects)	**High**
Enforce Task-Level CPU and Memory Resource Limits	Denial-of-Service (DoS), Noisy Neighbor	N/A	3.4 (Container Security)	**High**
Enable Runtime Threat Detection with GuardDuty	Malware Execution, Cryptomining, Post-Exploitation Activity	3.1 (Principle)	3.4 (Runtime Protection)	**Medium**
Encrypt Ephemeral Storage with Customer-Managed Keys (CMKs)	Unauthorized Data Access, Compliance Violations	2.7, 2.8	3.1.4 (Principle of Data Protection)	**Medium**

Control Name	Threat(s) Mitigated	CIS AWS Foundations v6.0.0 Mapping [121-124]	NIST SP 800-190 Mapping [125-129]	Priority
Utilize External and Dedicated Secret Stores	Secret Exposure, Credential Theft, Accidental Logging of Secrets	N/A	3.1.4 (Embedded Clear Text Secrets)	**High**
Enable Comprehensive Logging with Container Insights	Attacker Obfuscation, Lack of Forensic Data, Undetected Anomalies	2.1, 2.4, Section 3	3.1.2, 3.4 (Runtime Protection)	**High**
Monitor Network Traffic with VPC Flow Logs	Unauthorized Network Reconnaissance, Data Exfiltration, C2 Channels	2.9	3.3.3, 3.4 (Container Security)	**High**
Harden Base Images and Use Multi-Stage Builds	Vulnerability Exploitation, Increased Attack Surface	N/A	3.1 (Image Security), 3.1.2 (Configuration Defects)	**High**
Scan Images for Vulnerabilities and Enforce Signing	Deployment of Known Exploits, Image Tampering, Supply Chain Attacks	N/A	3.1.1 (Vulnerability Management), 3.1.5 (Untrusted Images)	**High**
Generate and Maintain a Software Bill of Materials (SBOM)	Lack of Supply Chain Visibility, Slow Response to Zero-Day Vulns	N/A	3.1 (Image Security), 3.1.1 (Vulnerability Management)	**Medium**

7. Implementation Guide

This guide provides practical, step-by-step instructions for implementing the mandatory security controls for AWS ECS Fargate. It includes examples for both Infrastructure as Code (IaC) using Terraform and manual configuration through the AWS Management Console. Following these instructions will help establish a secure and compliant Fargate environment.

1. Secure Task Definition

A secure task definition is the first line of defense, hardening the container's runtime environment [154]. Key controls include enforcing a read-only root filesystem, running as a non-root user, and setting strict resource limits. Making the filesystem immutable prevents attackers from modifying system files or installing malware, while running as a non-root user limits the impact of a process compromise [131].

Terraform Implementation This Terraform code defines an `aws_ecs_task_definition` that incorporates these security best practices [133]. It enforces a read-only root filesystem while providing a writable temporary directory, specifies a non-root user, and sets resource limits.

```
# Define a secure ECS Task Definition for a Fargate application.
resource "aws_ecs_task_definition" "secure_app_task_def" {
  family                   = "secure-app-family"
  requires_compatibilities = ["FARGATE"]
```

```
network_mode             = "awsvpc"
cpu                      = "1024" # 1 vCPU
memory                   = "2048" # 2 GB

# Reference the IAM roles created for task execution and application access.
# These roles are detailed in the "IAM Roles for Least Privilege" section.
execution_role_arn = aws_iam_role.ecs_task_execution_role.arn
task_role_arn      = aws_iam_role.ecs_task_role.arn

# Define a volume for temporary, writable storage.
# For Fargate, an empty 'host' block uses ephemeral storage.
volume {
  name = "tmp_volume"
}

# Define the primary application container with security best practices.
container_definitions = jsonencode([
  {
    name  = "secure-app-container"
    image = "${aws_ecr_repository.app_repo.repository_url}:latest"
    cpu   = 1024 # Allocate the full task CPU to this container.
    memory = 2048 # Allocate the full task memory to this container.

    # Security-critical parameter: Enforce a read-only root filesystem.
    # This prevents modification of the container's base image and system files.
    readonlyRootFilesystem = true

    # Security-critical parameter: Run the container as a non-root user.
    # The user '1000' should be created in the Dockerfile (e.g., RUN adduser --uid 1000 --disabled-pa
    user = "1000"

    # Mount the writable volume to the /tmp directory inside the container.
    # This allows the application to write temporary files without compromising the read-only root fi
    mountPoints = [
      {
        sourceVolume  = "tmp_volume"
        containerPath = "/tmp"
        readOnly      = false # This specific mount point is writable.
      }
    ]

    # Configure centralized logging to CloudWatch.
    # This prevents logs from being written to the container filesystem and aids in monitoring.
    logConfiguration = {
      logDriver = "awslogs"
      options = {
        "awslogs-group"         = "/ecs/secure-app"
        "awslogs-region"        = "us-east-1"
        "awslogs-stream-prefix" = "ecs"
      }
    }

    portMappings = [
      {
          containerPort = 8080
          protocol      = "tcp"
        }
      ]
    }
  ])
}
```

AWS Console UI Implementation

1. Navigate to the **Elastic Container Service (ECS)** dashboard in the AWS Management Console.
2. Select **Task Definitions** from the navigation pane and click **Create new task definition**.
3. Configure the basic details: provide a **Task definition family** name, select **AWS Fargate** as the launch type, and specify the **Task size** (CPU and Memory) [173].
4. Assign the appropriate **Task role** and **Task execution role**.
5. In the **Storage** section, click **Add volume**. Name it `tmp_volume` and select **Bind mount** as the type. Leave other fields blank for Fargate's ephemeral storage.
6. In the **Container details** section, provide a name and the image URI.
7. Scroll to the **Environment** subsection within the container configuration. Under **Security**, check the box for **Read-only root filesystem** and enter the non-root user ID (e.g., `1000`) in the **User** field [134].
8. Scroll to the **Storage** subsection. Under **Mount points**, select the `tmp_volume` you created, enter `/tmp` as the **Container path**, and ensure the **Read only** checkbox is *not* checked for this specific mount [130].
9. Under **Log collection**, ensure logging to Amazon CloudWatch is enabled and configured.
10. Review all settings and click **Create**.

2. IAM Roles for Least Privilege

Applying the principle of least privilege through the Task Role and Task Execution Role is fundamental to Fargate security [156]. The Task Execution Role grants the ECS agent permissions to manage the task lifecycle, while the Task Role grants the application code permissions to access other AWS services [137, 157]. These roles must be distinct and have narrowly scoped policies.

Terraform Implementation This Terraform code defines a least-privilege Task Execution Role (for pulling images and logs) and a Task Role (for application access to a specific S3 bucket) [138].

```
# --- Task Execution Role ---
# This role is used by the ECS agent to manage the task (pull images, get secrets, send logs).

resource "aws_iam_role" "ecs_task_execution_role" {
  name = "secure-app-task-execution-role"

  # Trust policy allowing the ECS tasks service to assume this role.
  assume_role_policy = jsonencode({
    Version   = "2012-10-17",
    Statement = [
      {
        Action    = "sts:AssumeRole",
        Effect    = "Allow",
        Principal = {
          Service = "ecs-tasks.amazonaws.com"
        }
      }
    ]
  })
}

# Inline policy defining the least-privilege permissions for the Task Execution Role.
resource "aws_iam_role_policy" "ecs_task_execution_policy" {
  name = "secure-app-task-execution-policy"
  role = aws_iam_role.ecs_task_execution_role.id

  policy = jsonencode({
    Version   = "2012-10-17",
    Statement = [
      # Required permissions for the ECS agent to pull container images from ECR.
      {
        Effect   = "Allow",
        Action   = [
          "ecr:GetAuthorizationToken",
          "ecr:BatchCheckLayerAvailability",
          "ecr:GetDownloadUrlForLayer",
          "ecr:BatchGetImage"
        ],
        Resource = "*" # ECR actions require a wildcard resource.
      },
      # Required permissions to send container logs to CloudWatch Logs.
      {
        Effect   = "Allow",
        Action   = [
          "logs:CreateLogStream",
          "logs:PutLogEvents"
        ],
        Resource = "arn:aws:logs:us-east-1:123456789012:log-group:/ecs/secure-app:*"
      }
    ]
  })
}

# --- Task Role ---
# This role is assumed by the application code inside the container to access other AWS services.

resource "aws_iam_role" "ecs_task_role" {
  name = "secure-app-task-role"

  # Trust policy allowing the ECS tasks service to assume this role.
  assume_role_policy = jsonencode({
    Version   = "2012-10-17",
    Statement = [
      {
        Action    = "sts:AssumeRole",
        Effect    = "Allow",
        Principal = {
          Service = "ecs-tasks.amazonaws.com"
        }
      }
    ]
  })
```

```
}

# Inline policy defining the least-privilege permissions for the application.
resource "aws_iam_role_policy" "ecs_task_policy" {
  name = "secure-app-task-policy"
  role = aws_iam_role.ecs_task_role.id

  policy = jsonencode({
    Version   = "2012-10-17",
    Statement = [
      # Security-critical parameter: Granting read-only access to a specific S3 bucket prefix.
      # This prevents the application from accessing other buckets or writing/deleting objects.
      {
        Effect   = "Allow",
        Action   = [
          "s3:GetObject",
          "s3:ListBucket"
        ],
        Resource = [
          "arn:aws:s3:::my-secure-app-bucket",
          "arn:aws:s3:::my-secure-app-bucket/data/*"
        ]
      }
    ]
  })
}
```

AWS Console UI Implementation

1. Navigate to the **IAM** service in the AWS Console.
2. Click **Roles**, then **Create role**.
3. For **Trusted entity type**, select **AWS service**. For **Use case**, choose **Elastic Container Service** and then **Elastic Container Service Task** [157].
4. On the **Add permissions** page, instead of using a managed policy, create a new custom policy by clicking **Create policy** [158].
5. Use the JSON editor to paste the least-privilege policy definition (e.g., the Task Execution Policy from the Terraform example, updated with your resource ARNs).
6. Name and create the policy.
7. Return to the role creation wizard, attach the newly created policy, and give the role a name (e.g., `secure-app-task-execution-role`).
8. Repeat this entire process to create the second role, the **Task Role**, using its specific least-privilege policy (e.g., the S3 access policy).
9. Finally, assign these two roles in the appropriate fields of your ECS Task Definition.

3. Network Security

A secure network design isolates Fargate tasks from the public internet and controls traffic flow using private subnets, VPC endpoints, and restrictive security groups [154, 163]. This prevents unauthorized access and lateral movement.

Terraform Implementation This code provisions a secure VPC with public and private subnets, essential VPC endpoints, and tightly scoped security groups for a Fargate service fronted by an ALB [140, 141].

```
# --- VPC and Subnets ---
resource "aws_vpc" "main" {
  cidr_block = "10.0.0.0/16"
  tags = { Name = "fargate-vpc" }
}

resource "aws_subnet" "private_subnet" {
  count             = 2
  vpc_id            = aws_vpc.main.id
  cidr_block        = "10.0.${count.index + 1}.0/24"
  availability_zone = "us-east-1${element(["a", "b"], count.index)}"
  tags = { Name = "fargate-private-subnet-${count.index}" }
}

resource "aws_subnet" "public_subnet" {
  count                   = 2
  vpc_id                  = aws_vpc.main.id
  cidr_block              = "10.0.10${count.index + 1}.0/24"
  availability_zone       = "us-east-1${element(["a", "b"], count.index)}"
  map_public_ip_on_launch = true
  tags = { Name = "fargate-public-subnet-${count.index}" }
}

# --- Internet Gateway for public subnets ---
resource "aws_internet_gateway" "igw" {
  vpc_id = aws_vpc.main.id
}
# ... (Route table and association for public subnets) ...

# --- VPC Endpoints for Private Connectivity ---
resource "aws_vpc_endpoint" "s3_gateway" {
  vpc_id       = aws_vpc.main.id
  service_name = "com.amazonaws.us-east-1.s3"
  vpc_endpoint_type = "Gateway"
  # ... (Route table association for private subnets) ...
}

resource "aws_vpc_endpoint" "ecr_dkr" {
  vpc_id              = aws_vpc.main.id
  service_name        = "com.amazonaws.us-east-1.ecr.dkr"
  vpc_endpoint_type   = "Interface"
  private_dns_enabled = true
  subnet_ids          = [for s in aws_subnet.private_subnet : s.id]
  security_group_ids  = [aws_security_group.vpc_endpoint_sg.id]
}
# ... (Additional endpoints for ecr.api, logs) ...

# --- Security Groups ---
resource "aws_security_group" "alb_sg" {
  name        = "fargate-alb-sg"
  description = "Allow HTTPS traffic to ALB"
  vpc_id      = aws_vpc.main.id
  ingress {
    from_port   = 443
    to_port     = 443
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }
  # ... (Egress rule) ...
}

resource "aws_security_group" "fargate_task_sg" {
  name        = "fargate-task-sg"
  description = "Allow traffic from ALB to Fargate task"
  vpc_id      = aws_vpc.main.id
  ingress {
    from_port       = 8080
    to_port         = 8080
    protocol        = "tcp"
    security_groups = [aws_security_group.alb_sg.id]
  }
  # ... (Egress rule to VPC endpoints) ...
}

resource "aws_security_group" "vpc_endpoint_sg" {
  name        = "fargate-vpc-endpoint-sg"
  description = "Allow traffic from Fargate tasks to VPC endpoints"
  vpc_id      = aws_vpc.main.id
  ingress {
    from_port       = 443
    to_port         = 443
    protocol        = "tcp"
    security_groups = [aws_security_group.fargate_task_sg.id]
  }
}

# --- ECS Service with Secure Network Configuration ---
resource "aws_ecs_service" "secure_app_service" {
  name            = "secure-app-service"
  # ... (cluster, task_definition) ...
  launch_type     = "FARGATE"

  network_configuration {
    subnets = [for s in aws_subnet.private_subnet : s.id]
    security_groups = [aws_security_group.fargate_task_sg.id]
    assign_public_ip = false
  }
  # ... (load_balancer configuration) ...
}
```

AWS Console UI Implementation

1. In the **VPC** console, use the **Create VPC** wizard and select **VPC and more**. Configure your VPC with public and private subnets across multiple AZs. Choose to create an S3 Gateway endpoint but select **0 NAT gateways** [163].
2. Navigate to **Endpoints** and create the necessary interface endpoints (`ecr.dkr`, `ecr.api`, `logs`), associating them with your private subnets and a dedicated security group [74].
3. Navigate to **EC2 > Security Groups** and create three groups:
   - `fargate-alb-sg`: Inbound rule for HTTPS (443) from `0.0.0.0/0`.
   - `fargate-task-sg`: Inbound rule for the application port (e.g., 8080) with the source set to the

  `fargate-alb-sg`'s ID.
  - `fargate-vpc-endpoint-sg`: Inbound rule for HTTPS (443) with the source set to the `fargate-task-sg`'s ID.

4. When creating the ECS service, on the **Networking** page, select your VPC, choose only the **private subnets**, select the `fargate-task-sg`, and ensure **Public IP** is turned **Off** [161, 162].

4. Encryption

Encrypting data at rest and in transit is a foundational security requirement. This involves using AWS KMS customer-managed keys (CMKs) to protect Fargate ephemeral storage, secrets in Secrets Manager, and data in EFS file systems [166].

Terraform Implementation This code creates a customer-managed KMS key and applies it to an ECS cluster's storage configuration and a secret in Secrets Manager [145, 147].

```
# --- Customer-Managed KMS Key ---
resource "aws_kms_key" "fargate_key" {
  description             = "KMS key for Fargate ephemeral storage and secrets"
  enable_key_rotation     = true
  deletion_window_in_days = 10
}

# --- ECS Cluster with Encrypted Ephemeral Storage ---
resource "aws_ecs_cluster" "main" {
  name = "secure-cluster"

  # Configure the cluster to use the CMK for encrypting Fargate task ephemeral storage.
  # This is configured via the execute_command_configuration.
  configuration {
    execute_command_configuration {
      kms_key_id = aws_kms_key.fargate_key.arn
      logging    = "OVERRIDE"
      log_configuration {
        cloud_watch_log_group_name = "/ecs/exec-logs"
      }
    }
  }
}

# --- Encrypted Secret in Secrets Manager ---
resource "aws_secretsmanager_secret" "db_credentials" {
  name = "my-app/db-credentials"
  # Encrypt the secret using the customer-managed KMS key.
  kms_key_id = aws_kms_key.fargate_key.arn
}

resource "aws_secretsmanager_secret_version" "db_credentials_version" {
  secret_id     = aws_secretsmanager_secret.db_credentials.id
  secret_string = "{\"username\":\"admin\",\"password\":\"supersecret\"}"
}
```

AWS Console UI Implementation

1. Navigate to the **Key Management Service (KMS)** console and create a new symmetric, encrypt/decrypt key. Define its alias, administrators, and usage permissions.

2. Navigate to the **ECS** console and select your cluster. To apply the key to ephemeral storage, you can configure the **Execute Command** setting on the cluster's configuration tab and select your newly created KMS key [166].
3. Navigate to **Secrets Manager** and click **Store a new secret**.
4. Follow the wizard to define the secret's contents. On the configuration page, in the **Encryption key** section, select your customer-managed KMS key from the dropdown menu before creating the secret [165].

5. Monitoring and Logging

Robust monitoring and logging are essential for security visibility. This involves enabling CloudWatch Container Insights for performance metrics, centralizing logs with the `awslogs` driver, and enabling GuardDuty for runtime threat detection [169].

Terraform Implementation This code enables Container Insights on the cluster, configures logging in the task definition, and enables GuardDuty ECS Runtime Monitoring [153].

```
# --- ECS Cluster with Container Insights Enabled ---
resource "aws_ecs_cluster" "main" {
  name = "secure-cluster"

  setting {
    name  = "containerInsights"
    value = "enabled"
  }
}

# --- Task Definition with Centralized Logging ---
resource "aws_ecs_task_definition" "secure_app_task_def" {
  # ... other parameters ...
  container_definitions = jsonencode([
    {
      # ... other container parameters ...
      logConfiguration = {
        logDriver = "awslogs"
        options = {
          "awslogs-group"         = "/ecs/secure-app"
          "awslogs-region"        = "us-east-1"
          "awslogs-stream-prefix" = "ecs"
        }
      }
    }
  ])
}

# --- GuardDuty for Runtime Threat Detection ---
data "aws_guardduty_detector" "current" {}

resource "aws_guardduty_detector_feature" "ecs_runtime_monitoring" {
  detector_id = data.aws_guardduty_detector.current.id
  name        = "ECS_FARGATE_AGENT_MANAGEMENT"
  status      = "ENABLED"
}

resource "aws_guardduty_detector_feature" "runtime_monitoring" {
  detector_id = data.aws_guardduty_detector.current.id
  name        = "RUNTIME_MONITORING"
  status      = "ENABLED"
  depends_on = [aws_guardduty_detector_feature.ecs_runtime_monitoring]
}
```

AWS Console UI Implementation

1. **Enable Container Insights:** In the **ECS** console, select your cluster, go to the **Configuration** tab, and edit the **Monitoring** settings to enable Container Insights [168].
2. **Configure Logging:** In your task definition's container configuration, ensure **Log collection** is enabled and set to **Amazon CloudWatch**.
3. **Enable GuardDuty:** In the **GuardDuty** console, navigate to **Runtime Monitoring** under Settings. Find the **ECS Fargate** section, click **Edit**, and toggle the status to **Enabled** [152].

6. ECR Image Security

Securing the container image involves automated vulnerability scanning, lifecycle management to remove old images, and restrictive repository policies to control access.

Terraform Implementation This code creates an ECR repository with scan-on-push enabled, a lifecycle policy, and a restrictive repository access policy.

```
# --- ECR Repository with Enhanced Scanning ---
resource "aws_ecr_repository" "app_repo" {
  name = "secure-app-repo"

  image_scanning_configuration {
    scan_on_push = true
  }
  image_tag_mutability = "IMMUTABLE"
}

# --- ECR Lifecycle Policy ---
resource "aws_ecr_lifecycle_policy" "app_repo_policy" {
  repository = aws_ecr_repository.app_repo.name
  policy = jsonencode({
    rules = [
      {
        rulePriority = 1,
        description  = "Expire untagged images after 14 days",
        selection    = { tagStatus = "untagged", countType = "sinceImagePushed", countUnit = "days", co
        action       = { type = "expire" }
      }
    ]
  })
}

# --- ECR Repository Policy ---
resource "aws_ecr_repository_policy" "repo_access_policy" {
  repository = aws_ecr_repository.app_repo.name
  policy = jsonencode({
    Version = "2012-10-17",
    Statement = [
      {
        Sid    = "AllowPullFromECSTaskExecutionRole",
        Effect = "Allow",
        Principal = { AWS = aws_iam_role.ecs_task_execution_role.arn },
        Action = [ "ecr:GetDownloadUrlForLayer", "ecr:BatchGetImage" ]
      }
      # Add another statement to allow push from your CI/CD role
    ]
  })
}
```

AWS Console UI Implementation

1. In the **ECR** console, create a new private repository. Set **Tag immutability** to **Immutable** and enable **Scan on push** [174].
2. After creation, select the repository and navigate to **Lifecycle Policy** in the left pane. Create rules to expire old or untagged images.
3. Navigate to **Permissions** and edit the policy JSON to restrict push and pull access to specific IAM roles.

7. Complete End-to-End Example

This final section provides a holistic Terraform configuration that integrates all the previously discussed security principles into a single, deployable Fargate service. It serves as a production-grade starting point.

Full Terraform Implementation This comprehensive code block defines a secure network, KMS key, ECR repository, IAM roles, a secure task definition, a load balancer, and an ECS service that ties everything together [74, 141, 143].

```
# ----------------------------------------------------------------
# Provider and Data Sources
# ----------------------------------------------------------------
provider "aws" {
  region = "us-east-1"
}

data "aws_availability_zones" "available" {}
data "aws_caller_identity" "current" {}

# ----------------------------------------------------------------
# Networking (VPC, Subnets, IGW, Security Groups)
# ----------------------------------------------------------------
resource "aws_vpc" "main" {
  cidr_block           = "10.10.0.0/16"
  enable_dns_support   = true
  enable_dns_hostnames = true
  tags = { Name = "secure-fargate-vpc" }
}

resource "aws_subnet" "public" {
  count                   = 2
  vpc_id                  = aws_vpc.main.id
  cidr_block              = "10.10.${100 + count.index}.0/24"
  availability_zone       = data.aws_availability_zones.available.names[count.index]
  map_public_ip_on_launch = true
  tags = { Name = "secure-fargate-public-${count.index}" }
```

```
}

resource "aws_subnet" "private" {
  count             = 2
  vpc_id            = aws_vpc.main.id
  cidr_block        = "10.10.${count.index}.0/24"
  availability_zone = data.aws_availability_zones.available.names[count.index]
  tags = { Name = "secure-fargate-private-${count.index}" }
}

resource "aws_internet_gateway" "igw" {
  vpc_id = aws_vpc.main.id
  tags = { Name = "secure-fargate-igw" }
}

resource "aws_route_table" "public" {
  vpc_id = aws_vpc.main.id
  route {
    cidr_block = "0.0.0.0/0"
    gateway_id = aws_internet_gateway.igw.id
  }
  tags = { Name = "secure-fargate-public-rt" }
}

resource "aws_route_table_association" "public" {
  count          = 2
  subnet_id      = aws_subnet.public[count.index].id
  route_table_id = aws_route_table.public.id
}

resource "aws_security_group" "alb_sg" {
  name   = "secure-fargate-alb-sg"
  vpc_id = aws_vpc.main.id
  ingress {
    from_port   = 443
    to_port     = 443
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }
  egress {
    from_port   = 0
    to_port     = 0
    protocol    = "-1"
    cidr_blocks = ["0.0.0.0/0"]
  }
}

resource "aws_security_group" "fargate_sg" {
  name   = "secure-fargate-task-sg"
  vpc_id = aws_vpc.main.id
  ingress {
    from_port       = 8080
    to_port         = 8080
    protocol        = "tcp"
    security_groups = [aws_security_group.alb_sg.id]
  }
  egress {
    from_port   = 0
    to_port     = 0
    protocol    = "-1"
    cidr_blocks = ["0.0.0.0/0"] # For a production system, this should be scoped to VPC endpoints or a N
  }
}

# ------------------------------------------------------------------
# KMS, ECR, IAM
# ------------------------------------------------------------------
resource "aws_kms_key" "fargate_key" {
  description         = "Key for secure Fargate app"
  enable_key_rotation = true
}

resource "aws_ecr_repository" "app" {
  name                 = "secure-fargate-app"
  image_tag_mutability = "IMMUTABLE"
  image_scanning_configuration {
    scan_on_push = true
  }
}

resource "aws_iam_role" "task_execution_role" {
  name               = "secure-fargate-execution-role"
  assume_role_policy = jsonencode({
    Version   = "2012-10-17",
    Statement = [{ Action = "sts:AssumeRole", Effect = "Allow", Principal = { Service = "ecs-tasks.amazo
  })
}

resource "aws_iam_role_policy_attachment" "task_execution_attachment" {
  role       = aws_iam_role.task_execution_role.name
  policy_arn = "arn:aws:iam::aws:policy/service-role/AmazonECSTaskExecutionRolePolicy"
}

resource "aws_iam_role" "task_role" {
  name               = "secure-fargate-task-role"
  assume_role_policy = jsonencode({
    Version   = "2012-10-17",
    Statement = [{ Action = "sts:AssumeRole", Effect = "Allow", Principal = { Service = "ecs-tasks.amazo
  })
}

# ------------------------------------------------------------------
# ECS Cluster, Task Definition, Service
# ------------------------------------------------------------------
resource "aws_ecs_cluster" "main" {
  name = "secure-fargate-cluster"
  setting {
    name  = "containerInsights"
    value = "enabled"
  }
}

resource "aws_ecs_task_definition" "app" {
  family                   = "secure-fargate-app"
  requires_compatibilities = ["FARGATE"]
  network_mode             = "awsvpc"
  cpu                      = "1024"
  memory                   = "2048"
  execution_role_arn       = aws_iam_role.task_execution_role.arn
  task_role_arn            = aws_iam_role.task_role.arn
  container_definitions = jsonencode([
    {
      name      = "app-container",
      image     = "${aws_ecr_repository.app.repository_url}:latest",
      readonlyRootFilesystem = true,
      user                   = "1000",
      logConfiguration = {
        logDriver = "awslogs",
        options = { "awslogs-group" = "/ecs/secure-fargate-app", "awslogs-region" = "us-east-1", "awslog
      },
      portMappings = [{ containerPort = 8080, protocol = "tcp" }]
    }
  ])
}

resource "aws_lb" "main" {
  name               = "secure-fargate-alb"
  load_balancer_type = "application"
  security_groups    = [aws_security_group.alb_sg.id]
  subnets            = [for s in aws_subnet.public : s.id]
}

resource "aws_lb_target_group" "app" {
  name        = "secure-fargate-tg"
  port        = 8080
  protocol    = "HTTP"
  vpc_id      = aws_vpc.main.id
  target_type = "ip"
  health_check { path = "/health" }
}

resource "aws_lb_listener" "https" {
  load_balancer_arn = aws_lb.main.arn
  port              = "443"
  protocol          = "HTTPS"
  ssl_policy        = "ELBSecurityPolicy-2016-08"
  certificate_arn   = "arn:aws:acm:us-east-1:${data.aws_caller_identity.current.account_id}:certificate
  default_action {
    type             = "forward"
    target_group_arn = aws_lb_target_group.app.arn
  }
}
```

```
resource "aws_ecs_service" "main" {
  name            = "secure-fargate-service"
  cluster         = aws_ecs_cluster.main.id
  task_definition = aws_ecs_task_definition.app.arn
  desired_count   = 2
  launch_type     = "FARGATE"
  network_configuration {
    subnets          = [for s in aws_subnet.private : s.id]
    security_groups  = [aws_security_group.fargate_sg.id]
    assign_public_ip = false
  }
  load_balancer {
    target_group_arn = aws_lb_target_group.app.arn
    container_name   = "app-container"
    container_port   = 8080
  }
  depends_on = [aws_lb_listener.https]
}
```

8. Audit Checklist

This checklist provides a systematic way to audit an AWS ECS Fargate environment against the mandatory security controls outlined in this baseline. Each item corresponds to a critical security configuration that must be verified.

Identity and Access Management

- ☐ Verify that the IAM Task Role and Task Execution Role are separate and distinct roles [100].
- ☐ Verify that the IAM Task Role policy grants only the minimum required permissions for the application to access other AWS services [157].
- ☐ Verify that the IAM Task Role policy avoids wildcards (`*`) for actions and resources.
- ☐ Verify that the IAM Task Execution Role policy is restricted to permissions for pulling images (ECR), sending logs (CloudWatch), and retrieving secrets (Secrets Manager/Parameter Store) [137].
- ☐ Verify that the trust policy for both roles only allows the `ecs-tasks.amazonaws.com` service principal to assume them.
- ☐ Verify that the trust policy for both roles includes `aws:SourceAccount` and `aws:SourceArn` conditions to prevent confused deputy vulnerabilities [159].

Networking

- ☐ Verify that all production Fargate tasks are launched in private subnets [164].
- ☐ Verify that the network configuration for all production Fargate services has `assignPublicIp` set to `DISABLED` [162].
- ☐ Verify that the VPC does not have an Internet Gateway attached if it is intended for fully internal workloads.
- ☐ Verify that VPC Endpoints are in use for all communication between Fargate tasks and supported AWS services (ECR, S3, Logs, Secrets Manager, etc.) [77].
- ☐ Verify that security groups for Fargate tasks have least-privilege inbound rules (e.g., only allowing traffic from the ALB security group on the application port).
- ☐ Verify that security groups for Fargate tasks have least-privilege outbound rules, restricting traffic to known destinations (e.g., VPC endpoints, database security groups).
- ☐ Verify that security group rules do not contain overly permissive CIDR ranges like `0.0.0.0/0` unless absolutely necessary and documented.

Compute and Task Definition

- ☐ Verify that all workloads are using the Fargate launch type, which inherently prohibits privileged containers [106].
- ☐ Verify that every task definition specifies explicit CPU and memory limits [173].
- ☐ Verify that the `readonlyRootFilesystem` parameter is set to `true` for all production containers in the task definition [134].
- ☐ Verify that a writable volume is only mounted to specific, non-critical paths (e.g., `/tmp`) if required [130].
- ☐ Verify that the `user` parameter is set in the container definition to run the process as a non-root user.
- ☐ Verify that AWS GuardDuty is enabled in the account.
- ☐ Verify that GuardDuty ECS Runtime Monitoring is enabled and that the `aws-gd-agent` sidecar is running in newly launched tasks [170].

Storage and Encryption

- ☐ Verify that Fargate ephemeral storage is configured to be encrypted with a Customer-Managed Key (CMK) from KMS for sensitive workloads [166].
- ☐ Verify that application secrets are not stored in container images, environment variables, or source code [167].
- ☐ Verify that all application secrets are stored in AWS Secrets Manager or Parameter Store (SecureString type) [167].
- ☐ Verify that secrets stored in Secrets Manager/Parameter Store are encrypted with a Customer-Managed Key (CMK) [165].
- ☐ Verify that any attached EFS file systems have encryption at rest enabled with a CMK.
- ☐ Verify that any attached EFS file systems have encryption in transit enabled.

Observability and Monitoring

- ☐ Verify that CloudWatch Container Insights is enabled on all production ECS clusters [171].
- ☐ Verify that all containers in the task definition are configured with the `awslogs` log driver to stream logs to CloudWatch Logs [169].
- ☐ Verify that VPC Flow Logging is enabled for all VPCs hosting Fargate workloads.
- ☐ Verify that VPC Flow Logs are being published to either CloudWatch Logs or an S3 bucket for analysis and retention.
- ☐ Verify that AWS CloudTrail is enabled for the account and is logging all management events.

Image Security

- ☐ Verify that container images are built from a minimal, approved base image (e.g., distroless, Alpine).
- ☐ Verify that Dockerfiles use multi-stage builds to exclude build tools from the final image.
- ☐ Verify that ECR repositories are configured with `image_tag_mutability` set to `IMMUTABLE`.
- ☐ Verify that ECR repositories have `scan_on_push` enabled for automated vulnerability scanning [174].
- ☐ Verify that a process or pipeline automation is in place to review scan results and block deployments with critical/high vulnerabilities.
- ☐ Verify that ECR lifecycle policies are in place to clean up old and untagged images.
- ☐ Verify that ECR repository policies are configured to restrict push/pull access to authorized IAM principals only.
- ☐ Verify that a Software Bill of Materials (SBOM) is generated for each image build and stored in an accessible location.
- ☐ Verify that an image signing and verification process is implemented in the CI/CD pipeline.

9. References

1. EC2 vs Fargate: Which is Right for You? - ProsperOps

2. AWS Fargate - Amazon Elastic Container Service
3. AWS EC2 vs ECS vs Fargate: Performance & Cost Analysis - StoneFly
4. AWS Fargate vs EC2: Which one to choose? - StormIT
5. ECS vs. Fargate: What's the difference? - cloudonaut
6. Shared responsibility model - Amazon Elastic Container Service
7. AWS Shared Responsibility Model - Architecting HIPAA Security and Compliance on Amazon EKS
8. AWS Fargate Security Overview Whitepaper - d1.awsstatic.com
9. What is the AWS Shared Responsibility Model? - Upwind
10. How does the AWS Shared Responsibility Model work? - Xebia
11. What's the difference between zero trust vs. defense in depth? - TechTarget
12. Zero Trust vs. Defense in Depth: Unpacking Modern IT Security - Cynet
13. Zero Trust vs. Least Privilege: What's the Difference? - Jamf Blog
14. Did Zero Trust Kill Defense-in-Depth or Has Defense-in-Depth Improved Zero Trust? - KRONTECH
15. Secure data with Zero Trust - Microsoft Learn
16. CIS AWS Foundations Benchmark - AWS Security Hub
17. Amazon Web Services - CISecurity.org
18. What are CIS Benchmarks? - AWS
19. Introducing CIS Amazon EKS Benchmark - AWS Containers Blog
20. Amazon Web Services Foundations Benchmark - AWS re:Post
21. SP 800-190, Application Container Security Guide - NIST Computer Security Resource Center
22. A guide to NIST compliance in container environments - Red Hat
23. NIST Special Publication 800-190 - nvlpubs.nist.gov
24. Draft SP 800-190, Application Container Security Guide - NIST Computer Security Resource Center
25. NIST 800-190 - Anchore
26. stride-gpt - GitHub
27. Threat modeling for container environments - ScienceDirect
28. Securing Container using Threat Modelling (STRIDE) - Medium
29. Container Infrastructure Threat Modelling - Medium
30. Threat Modeling and Security Analysis of Containers: A Survey - ResearchGate
31. Supply Chain Threats Using Container Images - Aqua Security Blog
32. Container Security Vulnerabilities & How to Mitigate Them - SentinelOne
33. Analysis of supply chain attacks through public Docker images - Sysdig
34. Base image vulnerabilities and risks in container security - Wiz
35. Threat Alert: Supply Chain Attacks Using Container Images to Target Cloud Native Environments - Global Security Magazine
36. Secrets - Kubernetes
37. Secrets as env vars - Reddit
38. Distribute Credentials Securely Using Secrets - Kubernetes
39. Using Kubernetes Secrets with environment variables and volume mounts - Mirantis
40. Secrets in Kubernetes: A Security Guide - Plural
41. Container Runtime Threat Hunting for SOC Analysts - Quzara
42. What is a Container Escape Vulnerability? - LinuxSecurity.com
43. runc Vulnerabilities Can be Exploited to Escape Containers - SecurityWeek
44. Leaky Vessels: Container escape vulnerabilities - Wiz Blog
45. Lateral movement in the cloud: from a container to the entire cluster - Sysdig
46. Security and IAM roles - Amazon Elastic Container Service
47. What are AWS IAM roles? - Wiz
48. IAM roles for tasks - Amazon Elastic Container Service
49. 15+ AWS IAM Best Practices - Spacelift
50. Best practices in IAM - AWS Identity and Access Management
51. Public facing web on ECS Fargate with CloudFormation - Containers on AWS
52. Build a Serverless Web Application on Fargate, ECS and CDK - The RantHe-Builder's Cloud
53. AWS Fargate - Aqua Security
54. Create and run web app on ECS using AWS Fargate - Medium

55. Building a secure HTTPS web server with Fargate, ACM, ALB - Stack Overflow
56. Secure your ALB with these best practices - learnaws.org
57. Security best practices when using ALB authentication - AWS Networking & Content Delivery Blog
58. AWS Fargate Application Load Balancer SSL Termination - Server Fault
59. A complete ECS Fargate setup with private subnets, NAT gateway, and ALB - GitHub Gist
60. AWS WAF AWS Managed Rule groups list - AWS WAF, AWS Firewall Manager, and AWS Shield Advanced
61. AWS Managed Rule groups - AWS WAF, AWS Firewall Manager, and AWS Shield Advanced
62. Rule groups - AWS WAF, AWS Firewall Manager, and AWS Shield Advanced
63. AWS WAF - Protect Your Web Applications From Common Exploits - GeekyAnts Blog
64. Managed rule groups - AWS WAF, AWS Firewall Manager, and AWS Shield Advanced
65. How to enable TLS v1.3 in ALB? - AWS re:Post
66. How to install certificate in ECS Fargate container to enable TLS all the way to the container? - Stack Overflow
67. Does AWS Application Load Balancer support TLS 1.3? - Stack Overflow
68. aws fargate application load balancer ssl termination - Server Fault
69. [ecs] [fargate] [service-connect]: End-to-end TLS 1.3 with ALB - GitHub
70. Selecting subnets for a service in Fargate - Server Fault
71. ECS Fargate: Why bother using private subnets + NAT gateway? - Reddit
72. How can I run Amazon ECS tasks on Fargate in a private subnet? - AWS re:Post
73. Fargate ECS cluster in public subnet - Reddit
74. Configuring ECS Fargate and ECR with Private Subnets - Tinfoil Cipher
75. Access container applications privately on Amazon ECS by using AWS Fargate, AWS PrivateLink, and a Network Load Balancer - AWS Prescriptive Guidance
76. Access private applications on AWS Fargate using Amazon API Gateway PrivateLink - AWS Compute Blog
77. VPC endpoints - Amazon Elastic Container Service
78. Network security - Amazon Elastic Container Service
79. mTLS Authentication with AWS ALB and ECS Fargate - copebit
80. Identity-Aware Proxy on ECS - omerxx.com
81. Three things to consider when implementing mutual TLS with AWS App Mesh - AWS Containers Blog
82. Transforming Istio into an enterprise-ready service mesh for Amazon ECS - AWS Containers Blog
83. Maintaining transport layer security all the way to your container using the Application Load Balancer with Amazon ECS and Envoy - AWS Containers Blog
84. Setup Fargate on private subnet without NAT - Reddit
85. Not able to run Fargate tasks in a private subnet - Reddit
86. ECS Cluster in an isolated VPC with no NAT Gateway - Containers on AWS
87. Fargate task networking - Amazon Elastic Container Service
88. How to to launch ECS Fargate container without public IP? - AWS re:Post
89. AWS App Mesh Features - Amazon Web Services
90. AWS App Mesh Frequently Asked Questions - Amazon Web Services
91. Using Service Meshes in AWS - d1.awsstatic.com
92. Using AWS App Mesh with Fargate | Amazon Web Services - AWS Compute Blog
93. aws-app-mesh-examples - GitHub
94. AWS Fargate can't resolve private DNS (Route 53) - Stack Overflow
95. How can I make the same Route 53 domain name resolve to an internal load balancer within the VPC and an external load balancer outside it? - AWS re:Post
96. ECS Fargate container cannot reach Route 53 URL - Stack Overflow
97. Assigning a domain name to an AWS Fargate task - Server Fault
98. Working with private hosted zones - Amazon Route 53
99. IAM for Amazon ECS on AWS Fargate - DEV Community
100. Task role vs. task execution role in Amazon ECS: A closer look at container security - Alexander Hose
101. IAM roles for Amazon ECS tasks - Amazon Elastic Container Service
102. How to to launch ecs fargate container without public ip - Stack Overflow

103. Enabling or Disabling Public IP for ECS Fargate Instances in AWS - Medium
104. How Runtime Monitoring works with Amazon ECS clusters on Fargate - AWS GuardDuty
105. Prerequisites for Runtime Monitoring support for Amazon ECS clusters - AWS GuardDuty
106. Fargate security considerations - Amazon Elastic Container Service
107. Fargate security considerations - Amazon Elastic Container Service
108. Securing Amazon ECS workloads on AWS Fargate with customer-managed keys - AWS Compute Blog
109. Data encryption and secrets management - Amazon EKS Best Practices Guides
110. Data - AWS EKS Best Practices Guide
111. Introducing server-side encryption of ephemeral storage using AWS Fargate-managed keys for AWS Fargate platform version 1.4 - AWS Containers Blog
112. Features - Amazon CloudWatch
113. Using Container Insights - Amazon CloudWatch
114. Amazon ECS CloudWatch Container Insights - Amazon Elastic Container Service
115. Amazon Elastic Container Service (ECS) with Container Insights for CloudWatch - Sumo Logic
116. Image security - AWS EKS Best Practices Guide
117. Cryptographic signing for containers - AWS Containers Blog
118. AWS Container Scanning: A Complete Guide - Wiz
119. Image scanning - Amazon ECR
120. Automating image compliance for Amazon EKS using Amazon Elastic Container Registry and AWS Security Hub - AWS Containers Blog
121. AWS_CIS_Foundations_Benchmark.pdf - d1.awsstatic.com
122. AWS_CIS_Foundations_Benchmark.pdf - d0.awsstatic.com
123. CIS AWS Foundations Benchmark controls - AWS Security Hub
124. Amazon Web Services (AWS) Benchmarks - cisecurity.org
125. NIST SP 800-190: Overview & Compliance Checklist - Anchore
126. NIST Special Publication 800-190 Application Container Security Guide - nvlpubs.nist.gov
127. Guide to NIST SP 800-190 compliance in container environments - Red Hat
128. NIST Special Publication (SP) 800-190, Application Container Security Guide - csrc.nist.gov
129. NIST 800-190 Checklist for Container Security - Aqua Security
130. I have to enable ECS readonlyRootFilesystem for security compliance, but my app needs to write to /tmp. What are my options? - reddit
131. How to set the container's root filesystem to read-only - DEV Community
132. aws ecs fargate enforce readonlyfilesystem - Stack Overflow
133. aws_ecs_task_definition - Terraform Registry
134. ecs-containers-readonly-access - AWS Config
135. Terraform and AWS IAM to update a secrets policy shared with multiple ECS tasks - Stack Overflow
136. Temporal on ECS Fargate with Terraform - papnori's blog
137. Task execution IAM role - Amazon ECS Developer Guide
138. 4.3 IAM Roles and Permissions - Terraform AWS ECS Module - DeepWiki
139. Attaching new permissions to role in AWS - Stack Overflow
140. umotif-public/ecs-fargate/aws - Terraform Registry
141. Deploying an ECS Fargate cluster with Terraform - finleap engineering
142. Issue creating aws_vpc_security_group within defined vpc - HashiCorp Discuss
143. Oxalide/terraform-fargate-example - GitHub
144. aws_ecs_cluster: support managedStorageConfiguration - GitHub
145. K8s and Terraform Secrets Encryption on AWS - Entro Security
146. aws_kms_secrets - Terraform Registry
147. How to use terraform to store a new secret in aws secrets manager using already created kms key? - Stack Overflow
148. How to enable secrets encryption on EKS cluster? - Stack Overflow
149. How to Detect Runtime Threats in Fargate Workloads - AWS for Engineers
150. Does ECS with Fargate support detailed monitoring? If so, how do I enable it for an existing cluster? - Stack Overflow
151. Container Insights - EKS Workshop

152. How Runtime Monitoring works with ECS (powered by Fargate) - AWS GuardDuty
153. cloudposse/terraform-aws-guardduty - GitHub
154. Security for tasks and containers - Amazon ECS Developer Guide
155. Security on AWS Fargate - Amazon ECS Developer Guide
156. ECS Fargate Security Best Practices - Medium
157. Amazon ECS task IAM role - Amazon ECS Developer Guide
158. Permissions required for Amazon ECS console - Amazon ECS Developer Guide
159. How do I configure my Amazon ECS task to assume an IAM role in another AWS account? - AWS re:Post
160. Amazon ECS container instance IAM role - Amazon ECS Developer Guide
161. Amazon ECS task networking for tasks on Fargate - Amazon ECS Developer Guide
162. How to to launch ECS Fargate container without public ip? - Stack Overflow
163. Setup AWS VPC for Fargate - Medium
164. Not able to run Fargate tasks in a private subnet - reddit
165. ECS Fargate service, who needs to access KMS for secrets? - Stack Overflow
166. Managing your AWS KMS keys for Fargate - Amazon ECS Developer Guide
167. How can I pass secrets or sensitive information securely to containers in an Amazon ECS task? - AWS re:Post
168. How to enable container insights on already created ecs fargate using aws console? - Stack Overflow
169. Monitoring your Amazon ECS containers - Amazon ECS Developer Guide
170. Troubleshooting GuardDuty in Amazon ECS - Amazon ECS Developer Guide
171. Using Container Insights - Amazon CloudWatch User Guide
172. AWS Fargate Metrics with CloudWatch - AWS for Engineers
173. Amazon ECS task definition parameters - Amazon ECS Developer Guide
174. Image scanning - Amazon ECR User Guide