API GATEWAY SECURITY

IMPLEMENTATION AN BEST PRACTICES

OKAN YILDIZ

AUGUST 2025

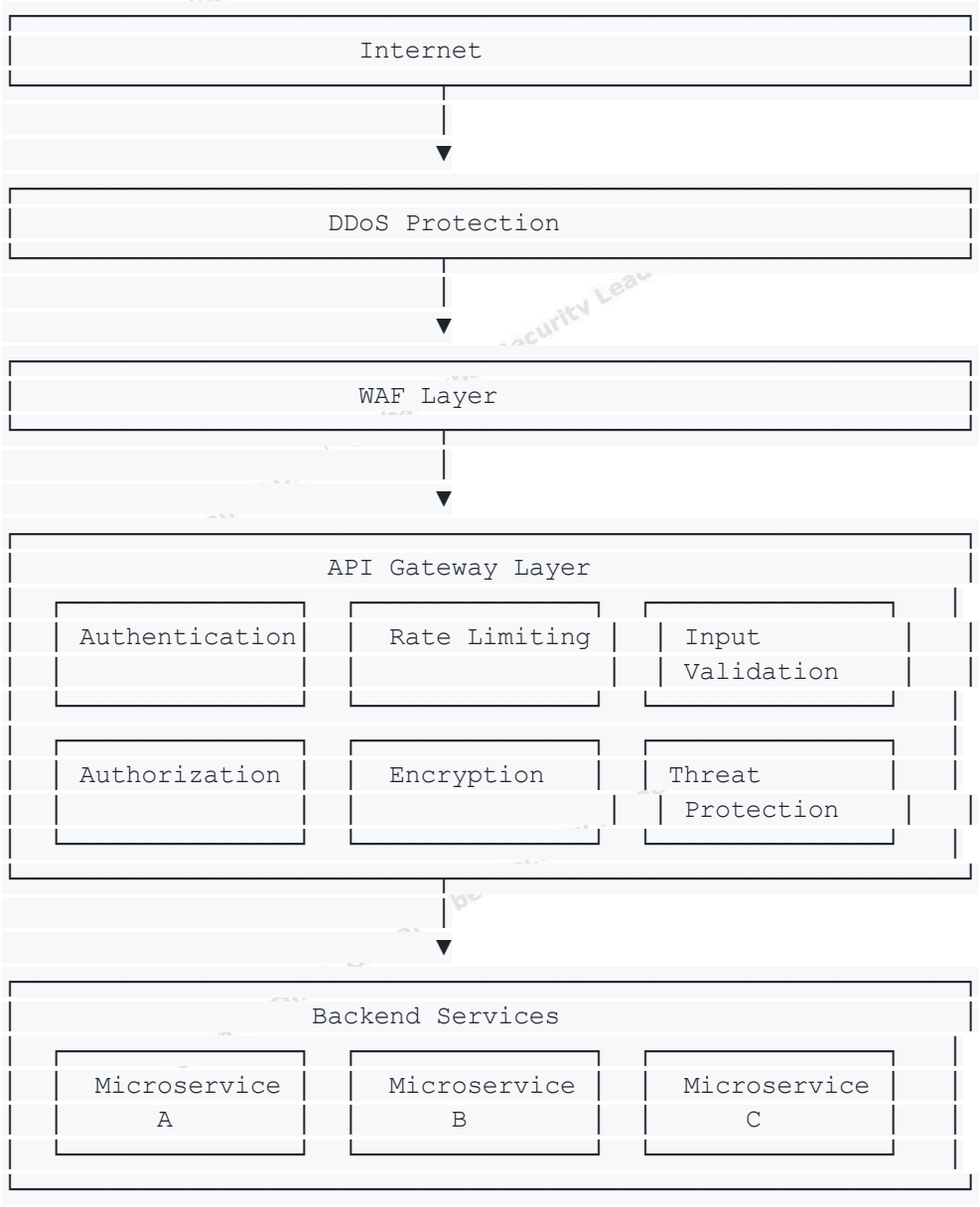
Introduction

In the modern distributed architecture landscape, API gateways have emerged as critical infrastructure components that function as the primary entry point for all API consumers. They serve as the centralized control plane, providing essential capabilities such as request routing, composition, protocol translation, authentication, and authorization. However, this strategic positioning also makes API gateways prime targets for attackers seeking to compromise backend services and data. This technical deep-dive explores comprehensive security strategies, implementation methods, and architectural considerations for hardening API gateways against sophisticated threats.

API Gateway Security Architecture

Understanding the API Gateway Security Perimeter

An effective API gateway security architecture implements multiple layers of defense to protect against various attack vectors:



Security Domains in API Gateway Architecture

To implement a secure API gateway, we must address multiple security domains:

1. Perimeter Security: Protection against network-level attacks

2. Transport Security: Encryption of data in transit
3. Access Control: Authentication and authorization mechanisms
4. API Abuse Protection: Rate limiting and anomaly detection
5. Data Protection: Input validation and output filtering
6. Operational Security: Logging, monitoring, and incident response

Essential Security Controls for API Gateways

1. Transport Layer Security (TLS)

Implementing strong TLS configuration is the foundation of API gateway security:

```
// Example Node.js/Express API Gateway TLS Configuration
const https = require('https');
const fs = require('fs');
const express = require('express');
const helmet = require('helmet');

const app = express();

// Apply security headers
app.use(helmet());

// TLS options with modern cipher suites and protocols
const tlsOptions = {
  key: fs.readFileSync('/path/to/private.key'),
  cert: fs.readFileSync('/path/to/certificate.crt'),
  ca: fs.readFileSync('/path/to/ca_bundle.crt'),
  minVersion: 'TLSv1.2',
  preferServerCipherSuites: true,
  cipherSuites: [
    'TLS_AES_256_GCM_SHA384',
    'TLS_CHACHA20_POLY1305_SHA256',
    'TLS_AES_128_GCM_SHA256',
    'ECDHE-ECDSA-AES256-GCM-SHA384',
    'ECDHE-RSA-AES256-GCM-SHA384',
    'ECDHE-ECDSA-CHACHA20-POLY1305',
    'ECDHE-RSA-CHACHA20-POLY1305',
    'ECDHE-ECDSA-AES128-GCM-SHA256',
    'ECDHE-RSA-AES128-GCM-SHA256'
  ],
  // Enable OCSP stapling
  requestOCSP: true
};

// Create HTTPS server with secure TLS configuration
const server = https.createServer(tlsOptions, app);
```

```
server.listen(443, () => {
  console.log('API Gateway running on port 443');
});
```

Best Practices for TLS:

- Enforce TLS 1.2+ with forward secrecy and strong cipher suites
- Implement automatic certificate rotation with short-lived certificates
- Use OCSP stapling to improve performance and privacy
- Implement HTTP Strict Transport Security (HSTS)
- Consider using mutual TLS (mTLS) for B2B integrations

2. Authentication and Identity Management

API gateways should implement robust, standardized authentication mechanisms:

```
// Example Spring Cloud Gateway Authentication Filter
@Component
public class JwtAuthenticationFilter implements GatewayFilter {

    private final JwtTokenValidator tokenValidator;
    private final ReactiveAuthenticationManager authManager;

    public JwtAuthenticationFilter(JwtTokenValidator tokenValidator,
                                   ReactiveAuthenticationManager authManager) {
        this.tokenValidator = tokenValidator;
        this.authManager = authManager;
    }

    @Override
    public Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain
chain) {
        ServerHttpRequest request = exchange.getRequest();

        // Extract JWT token from Authorization header
        List<String> authHeader = request.getHeaders().get("Authorization");

        if (authHeader != null && !authHeader.isEmpty()) {
            String authToken = authHeader.get(0);
            if (authToken.startsWith("Bearer ")) {
                String token = authToken.substring(7);

                // Validate token format, signature, expiration, etc.
                return tokenValidator.validateToken(token)
                    .flatMap(tokenDetails -> {
                        // Convert to Authentication object
```

```
                        JwtAuthenticationToken auth =
                            new JwtAuthenticationToken(token,
tokenDetails.getClaims());

                        // Authenticate through reactive authentication manager
                        return authManager.authenticate(auth)
                            .flatMap(authentication -> {
                                // Add authentication to the security context
                                SecurityContextImpl securityContext = new
SecurityContextImpl();

securityContext.setAuthentication(authentication);

                                // Modify request with authenticated user info
                                ServerHttpRequest mutatedRequest =
request.mutate()
                                    .header("X-User-ID",
tokenDetails.getSubject())
                                    .header("X-User-Roles", String.join(",",
tokenDetails.getRoles()))
                                    .build();

                                // Continue with modified request
                                return chain.filter(exchange.mutate()
                                    .request(mutatedRequest)
                                    .build());
                            });
                    })
                    .onErrorResume(error -> {
                        // Handle authentication errors

exchange.getResponse().setStatusCode(HttpStatus.UNAUTHORIZED);

exchange.getResponse().getHeaders().add("WWW-Authenticate",
                            "Bearer error=\"invalid_token\"");
                        return exchange.getResponse().setComplete();
                    });
            }
        }

        // No token provided - reject if this endpoint requires authentication
        if (isSecuredEndpoint(request.getPath().toString())) {
            exchange.getResponse().setStatusCode(HttpStatus.UNAUTHORIZED);
            exchange.getResponse().getHeaders().add("WWW-Authenticate",
"Bearer");
            return exchange.getResponse().setComplete();
        }

        // Endpoint doesn't require authentication
        return chain.filter(exchange);
    }
```

```
    private boolean isSecuredEndpoint(String path) {
        // Logic to determine if path requires authentication
        // ...
    }
}
```

Authentication Implementation Strategies:

1. JWT Authentication:
    - Use RS256 (RSA) rather than HS256 (HMAC) for signature verification
    - Enforce short token expiration times (15-60 minutes)
    - Implement token revocation capabilities
    - Validate token claims (iss, aud, exp, nbf, iat)
    - Store minimal information in tokens to reduce exposure
2. OAuth 2.0 and OpenID Connect:
    - Implement complete OAuth flow with separate authorization server
    - Use PKCE (Proof Key for Code Exchange) for public clients
    - Implement strict redirect_uri validation
    - Validate scopes against the requested resources
    - Use reference tokens for internal service communication
3. API Keys:
    - Use strong, randomly generated keys (min 32 bytes of entropy)
    - Store keys securely (hashed in database)
    - Implement key rotation capabilities
    - Associate keys with specific clients and permission sets
    - Never expose keys in URLs or logs

3. Authorization and Access Control

Granular authorization ensures authenticated users can only access appropriate resources:

```
# Example OPA (Open Policy Agent) Rego policy for API authorization
package httpapi.authz

# Default deny
default allow = false

# Allow GET access to /api/products for authenticated users with viewer role
allow {
    # Check request method
    input.method == "GET"

    # Check if path starts with /api/products
    startswith(input.path, "/api/products")
```

```
    # Check if user is authenticated
    input.user.authenticated == true

    # Check if user has viewer role
    contains(input.user.roles, "viewer")
}

# Allow all operations on /api/products for users with admin role
allow {
    # Check if path starts with /api/products
    startswith(input.path, "/api/products")

    # Check if user is authenticated
    input.user.authenticated == true

    # Check if user has admin role
    contains(input.user.roles, "admin")
}

# Allow specific user to access their own data
allow {
    # Check if path is a user-specific path
    startswith(input.path, "/api/users/")

    # Extract user ID from path
    path_parts := split(trim_prefix(input.path, "/api/users/"), "/")
    user_id := path_parts[0]

    # Check if user ID matches authenticated user
    input.user.id == user_id

    # Check if user is authenticated
    input.user.authenticated == true
}
```

Authorization Best Practices:

1. Implement Policy-Based Access Control:
    - Centralize authorization decisions
    - Use declarative policies rather than imperative code
    - Support attribute-based access control (ABAC)
    - Enable context-aware authorization (time, location, device)
2. Authorization Granularity:
    - Enforce authorization at multiple levels:
        - Service/API level
        - Resource level
        - Method level

        - Field level
    - Apply the principle of least privilege
3. Permission Propagation:
    - Pass minimal required authorization context to backends
    - Include user context in JWT or signed headers
    - Validate claims at both gateway and service levels

4. Rate Limiting and Traffic Control

Protecting APIs from abuse through sophisticated rate limiting is essential:

```
// Example Go code for advanced rate limiting in API Gateway
package ratelimit

import (
    "context"
    "net/http"
    "time"

    "github.com/redis/go-redis/v9"
)

// RateLimiter manages API rate limiting using Redis
type RateLimiter struct {
    redisClient *redis.Client
}

// RateLimitConfig defines rate limiting parameters
type RateLimitConfig struct {
    RequestsPerMinute int
    BurstSize         int
    ClientIdentifier  string   // IP, API key, user ID
    PathPatterns      []string // Which paths this config applies to
    Methods           []string // HTTP methods this config applies to
}

// NewRateLimiter creates a new rate limiter with Redis backend
func NewRateLimiter(redisURL string) (*RateLimiter, error) {
    options, err := redis.ParseURL(redisURL)
    if err != nil {
        return nil, err
    }

    client := redis.NewClient(options)
    return &RateLimiter{
        redisClient: client,
    }, nil
}
```

```go
// CheckRateLimit verifies if the request is within rate limits
func (rl *RateLimiter) CheckRateLimit(ctx context.Context, config
RateLimitConfig) (bool, int, time.Time, error) {
    // Use sliding window algorithm for rate limiting
    now := time.Now()
    windowKey := config.ClientIdentifier + ":" + now.Format("2006-01-02-15-04")

    // Create a transaction
    pipe := rl.redisClient.TxPipeline()

    // Increment the counter for this minute
    countResult := pipe.Incr(ctx, windowKey)

    // Set expiration for automatic cleanup
    pipe.Expire(ctx, windowKey, 2*time.Minute)

    // Execute transaction
    _, err := pipe.Exec(ctx)
    if err != nil {
        return false, 0, time.Time{}, err
    }

    // Get the current count
    count, err := countResult.Result()
    if err != nil {
        return false, 0, time.Time{}, err
    }

    // Calculate when the client can make another request
    resetTime := now.Truncate(time.Minute).Add(time.Minute)

    // Check if we've exceeded the rate limit
    if count > int64(config.RequestsPerMinute) {
        // Rate limit exceeded
        return false, int(count), resetTime, nil
    }

    // Request is allowed
    return true, int(count), resetTime, nil
}

// Middleware implements HTTP middleware for rate limiting
func (rl *RateLimiter) Middleware(config RateLimitConfig) func(http.Handler)
http.Handler {
    return func(next http.Handler) http.Handler {
        return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
            // Determine client identifier (IP, API key, etc.)
            clientID := getClientIdentifier(r, config.ClientIdentifier)

            // Check if this path and method should be rate limited
            if !shouldRateLimit(r, config) {
```

```
                next.ServeHTTP(w, r)
                return
            }

            // Check rate limit
            allowed, current, reset, err := rl.CheckRateLimit(r.Context(),
config)
            if err != nil {
                // In case of errors, allow the request to proceed
                // but log the error
                logRateLimitError(err)
                next.ServeHTTP(w, r)
                return
            }

            // Set rate limit headers
            w.Header().Set("X-RateLimit-Limit", fmt.Sprintf("%d",
config.RequestsPerMinute))
            w.Header().Set("X-RateLimit-Remaining", fmt.Sprintf("%d",
config.RequestsPerMinute-current))
            w.Header().Set("X-RateLimit-Reset", fmt.Sprintf("%d", reset.Unix()))

            if !allowed {
                // Rate limit exceeded
                w.Header().Set("Retry-After", fmt.Sprintf("%d",
int(reset.Sub(time.Now()).Seconds())))
                http.Error(w, "Rate limit exceeded", http.StatusTooManyRequests)
                return
            }

            // Proceed with the request
            next.ServeHTTP(w, r)
        })
    }
}

// Helper functions
func getClientIdentifier(r *http.Request, identifierType string) string {
    // Implementation to get client identifier based on configuration
    // ...
}

func shouldRateLimit(r *http.Request, config RateLimitConfig) bool {
    // Implementation to determine if request should be rate limited
    // ...
}

func logRateLimitError(err error) {
    // Implementation to log rate limiting errors
    // ...
}
```

Advanced Rate Limiting Strategies:

1. Multi-Dimensional Rate Limiting:
    - Client IP-based limits (prevent network-level abuse)
    - API key or user-based limits (fair resource allocation)
    - Endpoint-specific limits (protect vulnerable endpoints)
    - Global limits (protect overall system resources)
2. Adaptive Rate Limiting:
    - Dynamic adjustment based on backend capacity
    - Automatic throttling during traffic spikes
    - Custom rules for different client tiers
3. Rate Limit Response Handling:
    - Implement proper 429 (Too Many Requests) responses
    - Include Retry-After headers
    - Provide clear rate limit headers:
        - X-RateLimit-Limit
        - X-RateLimit-Remaining
        - X-RateLimit-Reset

5. Input Validation and Threat Protection

Robust input validation at the gateway level prevents many attacks:

```
// Example TypeScript validation middleware for API Gateway
import { Request, Response, NextFunction } from 'express';
import { AJV, JSONSchemaType } from 'ajv';
import { LRUCache } from 'lru-cache';

// Define interface for request data
interface ProductRequest {
  name: string;
  description: string;
  price: number;
  category: string;
  tags: string[];
}

// JSON Schema for validation
const productSchema: JSONSchemaType<ProductRequest> = {
  type: 'object',
  properties: {
    name: { type: 'string', minLength: 1, maxLength: 100 },
    description: { type: 'string', maxLength: 2000 },
    price: { type: 'number', minimum: 0 },
    category: { type: 'string', enum: ['electronics', 'clothing', 'food',
'other'] },
```

```
    tags: {
      type: 'array',
      items: { type: 'string', pattern: '^[a-zA-Z0-9-_]+$', maxLength: 50 },
      maxItems: 10
    }
  },
  required: ['name', 'price', 'category'],
  additionalProperties: false
};

// Instantiate validator
const ajv = new AJV({ allErrors: true, removeAdditional: 'all' });
const validate = ajv.compile(productSchema);

// Cache for compiled schemas
const schemaCache = new LRUCache<string, ReturnType<typeof ajv.compile>>({
  max: 100,
  ttl: 1000 * 60 * 60, // 1 hour
});

// Validation middleware
export function validateRequest(schemaName: string, schema: any) {
  // Get or compile validator
  let validator = schemaCache.get(schemaName);
  if (!validator) {
    validator = ajv.compile(schema);
    schemaCache.set(schemaName, validator);
  }

  return (req: Request, res: Response, next: NextFunction) => {
    // Decide what to validate based on request type
    const dataToValidate = req.method === 'GET' ? req.query : req.body;

    // Perform validation
    const valid = validator(dataToValidate);

    if (!valid) {
      // Format validation errors
      const errors = validator.errors?.map(err => ({
        field: err.instancePath,
        message: err.message
      }));

      return res.status(400).json({
        error: 'Invalid request data',
        details: errors
      });
    }

    // Request is valid, continue
    next();
```

```
  };
}

// Sanitization middleware for additional protection
export function sanitizeRequest() {
  return (req: Request, res: Response, next: NextFunction) => {
    // Sanitize headers
    const sanitizedHeaders = new Map<string, string>();

    for (const [key, value] of Object.entries(req.headers)) {
      // Remove any headers that might contain dangerous content
      if (key.toLowerCase().startsWith('x-forwarded-')) {
        // Only allow from trusted proxies
        if (isTrustedProxy(req.socket.remoteAddress)) {
          sanitizedHeaders.set(key, Array.isArray(value) ? value[0] : value as
string);
        }
      } else {
        sanitizedHeaders.set(key, Array.isArray(value) ? value[0] : value as
string);
      }
    }

    // Replace req.headers with sanitized headers
    req.headers = sanitizedHeaders as any;

    // If using Express raw body parser, sanitize the raw body too
    if (req.body && typeof req.body === 'string') {
      req.body = sanitizeString(req.body);
    }

    next();
  };
}

// Helper functions
function isTrustedProxy(ip: string | undefined): boolean {
  // Implementation to check if IP is a trusted proxy
  // ...
}

function sanitizeString(input: string): string {
  // Implementation to sanitize string (remove control chars, etc.)
  // ...
}
```

Input Validation Security Guidelines:

1. Schema-Based Validation:
    - Validate all request parameters, headers, and body content

    - Define and enforce strict schemas for all API operations
    - Implement white-listing rather than black-listing
    - Apply appropriate data type validation
2. Advanced Threat Protection:
    - SQL injection detection and prevention
    - XSS payload detection
    - Command injection prevention
    - XML/JSON entity attack mitigation
    - Regular expression denial of service (ReDoS) protection
3. Content Security:
    - Enforce maximum request sizes
    - Validate Content-Type headers
    - Implement deep inspection of complex structures
    - Sanitize output to prevent data leakage

6. API Gateway Logging and Monitoring

Comprehensive security monitoring is crucial for detecting and responding to attacks:

```
# Example OpenTelemetry configuration for API Gateway monitoring
receivers:
  otlp:
    protocols:
      grpc:
        endpoint: 0.0.0.0:4317
      http:
        endpoint: 0.0.0.0:4318

processors:
  batch:
    timeout: 1s
    send_batch_size: 1024

  # Filter sensitive data
  attributes/filter:
    actions:
      - key: http.request.header.authorization
        action: update
        value: "[REDACTED]"
      - key: http.request.header.cookie
        action: update
        value: "[REDACTED]"

  # Add security context
  attributes/security:
    actions:
      - key: security.threat_detected
```

```
        action: insert
        value: false
      - key: security.auth_status
        action: insert
        value: "unknown"

  # Resource detection
  resourcedetection:
    detectors: [env, system]
    timeout: 5s

exporters:
  elasticsearch:
    endpoints: ["https://elasticsearch:9200"]
    index: "api-gateway-logs"

  prometheus:
    endpoint: 0.0.0.0:8889

  otlp:
    endpoint: collector:4317
    tls:
      insecure: false
      cert_file: /certs/client.crt
      key_file: /certs/client.key
      ca_file: /certs/ca.crt

service:
  pipelines:
    traces:
      receivers: [otlp]
      processors: [attributes/filter, attributes/security, resourcedetection,
batch]
      exporters: [elasticsearch, otlp]

    metrics:
      receivers: [otlp]
      processors: [batch]
      exporters: [prometheus, otlp]

    logs:
      receivers: [otlp]
      processors: [attributes/filter, resourcedetection, batch]
      exporters: [elasticsearch, otlp]

  telemetry:
    logs:
      level: info
```

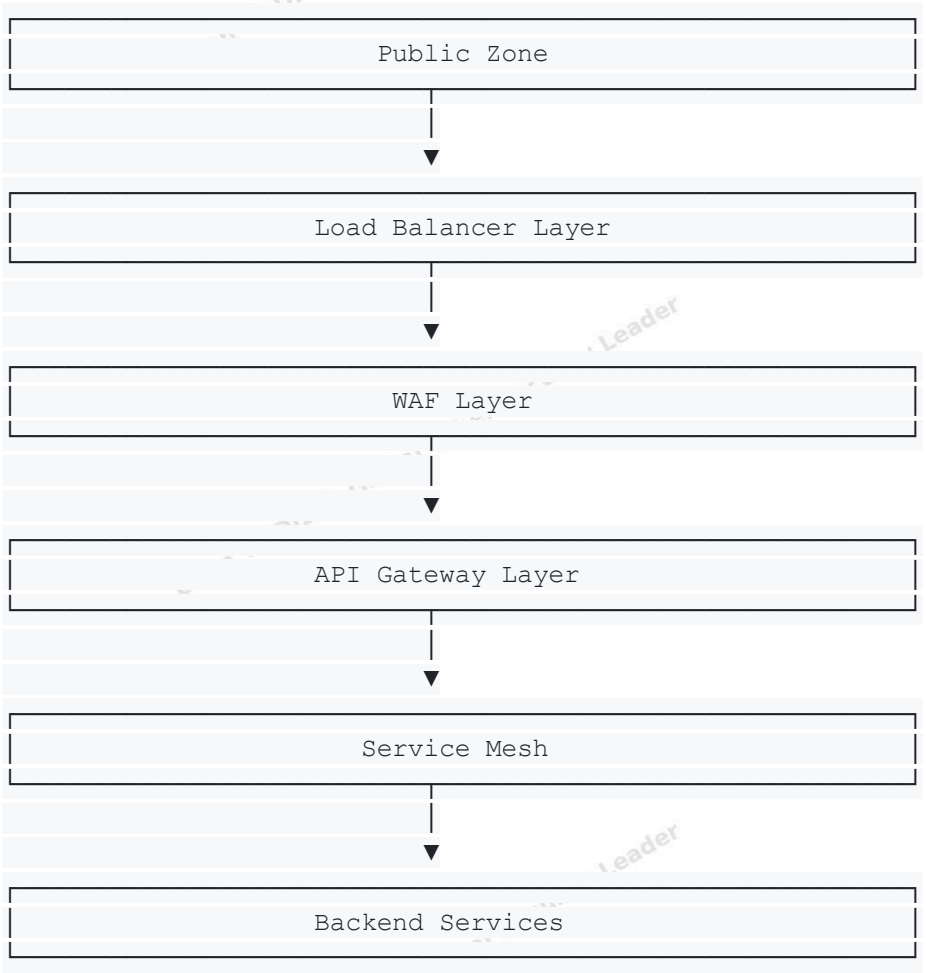
Security Monitoring and Logging Requirements:

1. Comprehensive Logging:
    - Log all API gateway events including:
        - Authentication attempts (success and failure)
        - Authorization decisions
        - Rate limiting actions
        - Security violations
        - Input validation failures
    - Include correlation IDs for request tracing
    - Encrypt sensitive log data
2. Real-Time Monitoring:
    - Implement anomaly detection for:
        - Request volume and patterns
        - Error rates
        - Authentication failures
        - Unexpected API usage patterns
    - Set up alerting for security-related events
    - Monitor response times as indicators of potential DoS
3. Security Analytics:
    - Aggregate logs for security analysis
    - Implement behavior-based threat detection
    - Correlate events across infrastructure
    - Maintain audit trails for compliance

API Gateway Deployment Security

Secure Deployment Architecture

Implement defense-in-depth with multiple security layers:



Secure Deployment Practices:

1. Infrastructure Security:
    - Use immutable infrastructure patterns
    - Implement strict network segmentation
    - Apply the principle of least privilege to all components
    - Protect API gateway configuration as sensitive data
    - Implement automated security testing in CI/CD
2. Container and Orchestration Security:
    - Run containers with non-root users
    - Implement runtime protection for containers
    - Use trusted base images with minimal attack surface
    - Apply network policies to restrict container communication
    - Implement secrets management

3. Cloud-Specific Controls:
    - Use cloud provider security services (AWS WAF, Azure Front Door, etc.)
    - Implement private endpoints for backend connectivity
    - Enable DDoS protection services
    - Apply appropriate IAM controls

Secure API Gateway Configuration

```
# Example Terraform configuration for secure API Gateway in AWS
resource "aws_api_gateway_rest_api" "secure_api" {
  name        = "secure-api-gateway"
  description = "Secure API Gateway with comprehensive security controls"

  endpoint_configuration {
    types = ["REGIONAL"]
  }

  # Enable detailed CloudWatch metrics
  minimum_compression_size = 1024

  policy = jsonencode({
    Version = "2012-10-17"
    Statement = [
      {
        Effect = "Allow"
        Principal = "*"
        Action = "execute-api:Invoke"
        Resource = "execute-api:/*"
        Condition = {
          IpAddress = {
            "aws:SourceIp" = var.allowed_ip_ranges
          }
        }
      }
    ]
  })
}

# Configure API Gateway WAF
resource "aws_wafv2_web_acl" "api_waf" {
  name        = "api-gateway-waf"
  description = "WAF for API Gateway"
  scope       = "REGIONAL"

  default_action {
    allow {}
  }
```

```
  # SQL injection prevention
  rule {
    name     = "SQLInjectionRule"
    priority = 1

    statement {
      sql_injection_match_statement {
        field_to_match {
          all_query_arguments {}
        }
        text_transformation {
          priority = 1
          type     = "URL_DECODE"
        }
      }
    }

    action {
      block {}
    }

    visibility_config {
      cloudwatch_metrics_enabled = true
      metric_name                = "SQLInjectionRule"
      sampled_requests_enabled   = true
    }
  }

  # Rate-based rule (IP-based rate limiting)
  rule {
    name     = "RateLimitRule"
    priority = 2

    statement {
      rate_based_statement {
        limit              = 1000
        aggregate_key_type = "IP"
      }
    }

    action {
      block {}
    }

    visibility_config {
      cloudwatch_metrics_enabled = true
      metric_name                = "RateLimitRule"
      sampled_requests_enabled   = true
    }
  }
```

```
  # Prevent XSS attacks
  rule {
    name     = "XSSRule"
    priority = 3

    statement {
      xss_match_statement {
        field_to_match {
          body {}
        }
        text_transformation {
          priority = 1
          type     = "HTML_ENTITY_DECODE"
        }
      }
    }

    action {
      block {}
    }

    visibility_config {
      cloudwatch_metrics_enabled = true
      metric_name                = "XSSRule"
      sampled_requests_enabled   = true
    }
  }

  visibility_config {
    cloudwatch_metrics_enabled = true
    metric_name                = "ApiGatewayWaf"
    sampled_requests_enabled   = true
  }
}

# Associate WAF with API Gateway stage
resource "aws_wafv2_web_acl_association" "api_waf_assoc" {
  resource_arn = aws_api_gateway_stage.production.arn
  web_acl_arn  = aws_wafv2_web_acl.api_waf.arn
}

# Configure secure API Gateway stage
resource "aws_api_gateway_stage" "production" {
  deployment_id = aws_api_gateway_deployment.latest.id
  rest_api_id   = aws_api_gateway_rest_api.secure_api.id
  stage_name    = "production"

  # Enable detailed CloudWatch logs
  access_log_settings {
    destination_arn = aws_cloudwatch_log_group.api_gateway.arn
    format = jsonencode({
```

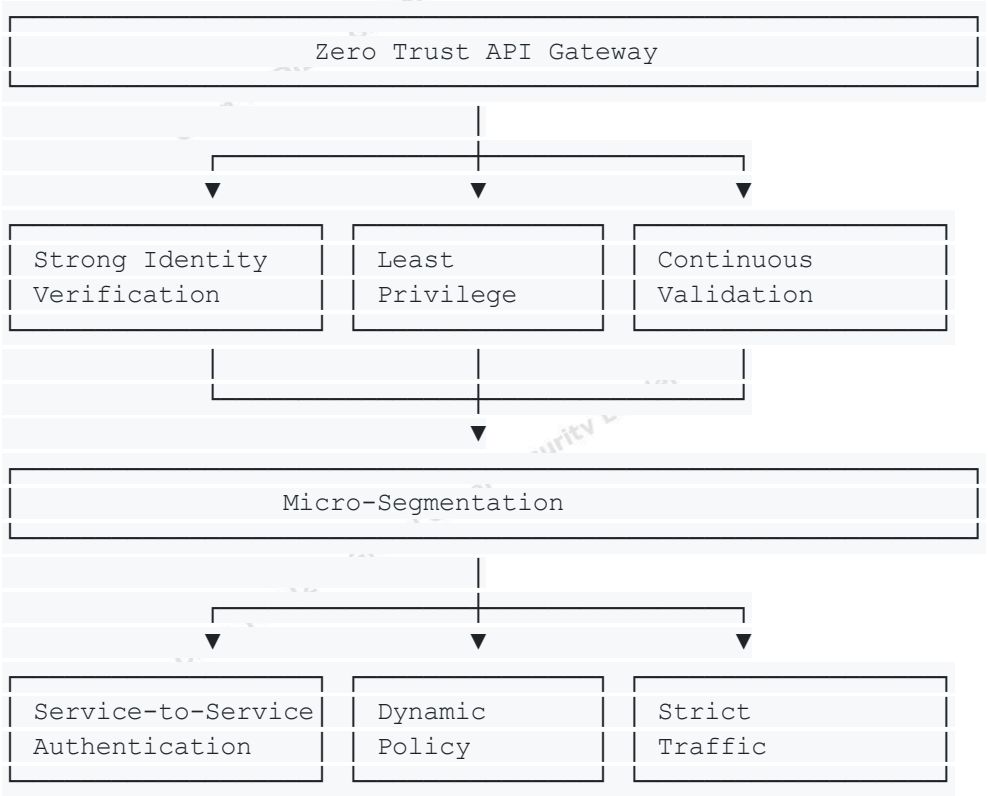
```
      requestId               = "$context.requestId"
      ip                      = "$context.identity.sourceIp"
      requestTime             = "$context.requestTime"
      httpMethod              = "$context.httpMethod"
      resourcePath            = "$context.resourcePath"
      status                  = "$context.status"
      protocol                = "$context.protocol"
      responseLength          = "$context.responseLength"
      integrationError        = "$context.integrationErrorMessage"
      authorizerError         = "$context.authorizer.error"
      authorizerLatency       = "$context.authorizer.latency"
      authorizerStatus        = "$context.authorizer.status"
      userAgent               = "$context.identity.userAgent"
      apiKeyId                = "$context.identity.apiKeyId"
      cognitoAuthenticationType = "$context.identity.cognitoAuthenticationType"
      cognitoIdentityId       = "$context.identity.cognitoIdentityId"
      cognitoIdentityPoolId   = "$context.identity.cognitoIdentityPoolId"
    })
  }

  # Enable X-Ray tracing
  xray_tracing_enabled = true

  # Configure throttling for all methods
  throttling_burst_limit = 1000
  throttling_rate_limit  = 500
}

# Configure secure gateway methods
resource "aws_api_gateway_method" "secure_method" {
  rest_api_id   = aws_api_gateway_rest_api.secure_api.id
  resource_id   = aws_api_gateway_resource.example.id
  http_method   = "POST"
  authorization = "COGNITO_USER_POOLS"
  authorizer_id = aws_api_gateway_authorizer.cognito.id

  # Require API key
  api_key_required = true

  # Request validator
  request_validator_id = aws_api_gateway_request_validator.full_validator.id

  # Validate request body
  request_models = {
    "application/json" = aws_api_gateway_model.example_model.name
  }
}

# Define request validator
resource "aws_api_gateway_request_validator" "full_validator" {
  name                        = "full-validator"
```

```
  rest_api_id                 = aws_api_gateway_rest_api.secure_api.id
  validate_request_body       = true
  validate_request_parameters = true
}

# Define API Gateway method responses with secure headers
resource "aws_api_gateway_method_response" "secure_response" {
  rest_api_id = aws_api_gateway_rest_api.secure_api.id
  resource_id = aws_api_gateway_resource.example.id
  http_method = aws_api_gateway_method.secure_method.http_method
  status_code = "200"

  # Add security headers
  response_parameters = {
    "method.response.header.X-Content-Type-Options" = true
    "method.response.header.X-Frame-Options" = true
    "method.response.header.Content-Security-Policy" = true
    "method.response.header.Strict-Transport-Security" = true
    "method.response.header.Cache-Control" = true
  }

  response_models = {
    "application/json" = aws_api_gateway_model.response_model.name
  }
}
```

Advanced API Gateway Security Strategies

Zero Trust Security Model for API Gateways

A Zero Trust architecture assumes no implicit trust, regardless of network location or asset ownership. For API gateways, this means:



Key Zero Trust Implementation Strategies for API Gateways:

1. Strong Identity Verification:
    - Implement mutual TLS (mTLS) between clients and gateways
    - Use strong authentication mechanisms (OAuth 2.0, OIDC)
    - Validate client certificates against trusted CAs
    - Enforce regular credential rotation
2. Continuous Authorization and Validation:
    - Re-validate tokens on each request
    - Implement real-time token revocation checks
    - Apply context-aware access policies
    - Evaluate risk signals dynamically
    - Never trust, always verify
3. Micro-Segmentation:
    - Isolate API gateway components from each other
    - Apply strict network controls between services
    - Implement service mesh for internal communication
    - Use dedicated gateways for different security domains
4. Monitoring and Analytics:

- Log all access attempts
- Continuously analyze traffic patterns
- Implement behavioral analytics
- Enable real-time threat detection

Service Mesh Integration with API Gateways

Modern architectures often combine API gateways with service meshes for comprehensive security:

```
// Example Envoy proxy configuration (for use in service mesh with API Gateway)
static_resources:
  listeners:
  - name: ingress_listener
    address:
      socket_address:
        address: 0.0.0.0
        port_value: 9000
    filter_chains:
    - filters:
      - name: envoy.filters.network.http_connection_manager
        typed_config:
          "@type":
type.googleapis.com/envoy.extensions.filters.network.http_connection_manager.v3.
HttpConnectionManager
          stat_prefix: ingress_http
          access_log:
          - name: envoy.access_loggers.file
            typed_config:
              "@type":
type.googleapis.com/envoy.extensions.access_loggers.file.v3.FileAccessLog
              path: /dev/stdout
              format: |
                [%START_TIME%] "%REQ(:METHOD)%
%REQ(X-ENVOY-ORIGINAL-PATH?:PATH)% %PROTOCOL%"
                %RESPONSE_CODE% %RESPONSE_FLAGS% %RESPONSE_CODE_DETAILS%
%CONNECTION_TERMINATION_DETAILS%
                "%UPSTREAM_TRANSPORT_FAILURE_REASON%" %BYTES_RECEIVED%
%BYTES_SENT% %DURATION%
                %RESP(X-ENVOY-UPSTREAM-SERVICE-TIME)% "%REQ(X-FORWARDED-FOR)%"
"%REQ(USER-AGENT)%"
                "%REQ(X-REQUEST-ID)%" "%REQ(:AUTHORITY)%" "%UPSTREAM_HOST%"
%UPSTREAM_CLUSTER%
                %UPSTREAM_LOCAL_ADDRESS% %DOWNSTREAM_LOCAL_ADDRESS%
%DOWNSTREAM_REMOTE_ADDRESS%
                %REQUESTED_SERVER_NAME% %ROUTE_NAME%
          http_filters:
          - name: envoy.filters.http.jwt_authn
            typed_config:
```

```
              "@type":
type.googleapis.com/envoy.extensions.filters.http.jwt_authn.v3.JwtAuthentication
              providers:
                primary_jwt:
                  issuer: https://auth.example.com
                  audiences:
                  - api.example.com
                  remote_jwks:
                    http_uri:
                      uri: https://auth.example.com/.well-known/jwks.json
                      timeout: 5s
                      cluster: jwks_cluster
                  from_headers:
                  - name: Authorization
                    value_prefix: "Bearer "
                  forward: true
                  payload_in_metadata: jwt_payload
              rules:
              - match:
                  prefix: /api/
                requires:
                  provider_name: primary_jwt
          - name: envoy.filters.http.rbac
            typed_config:
              "@type":
type.googleapis.com/envoy.extensions.filters.http.rbac.v3.RBAC
              rules:
                action: ALLOW
                policies:
                  product-api-read:
                    permissions:
                    - and_rules:
                        rules:
                        - header:
                            name: ":method"
                            exact_match: "GET"
                        - url_path:
                            path:
                              prefix: "/api/products"
                    principals:
                    - metadata:
                        filter: envoy.filters.http.jwt_authn
                        path:
                        - key: jwt_payload
                        - key: scope
                        value:
                          list_match:
                            one_of:
                              string_match:
                                exact: "product:read"
          - name: envoy.filters.http.ext_authz
```

```
            typed_config:
              "@type":
type.googleapis.com/envoy.extensions.filters.http.ext_authz.v3.ExtAuthz
              transport_api_version: V3
              with_request_body:
                max_request_bytes: 8192
                allow_partial_message: true
              failure_mode_allow: false
              grpc_service:
                google_grpc:
                  target_uri: ext-authz:9000
                  stat_prefix: ext_authz
                timeout: 0.5s
          - name: envoy.filters.http.router
            typed_config:
              "@type":
type.googleapis.com/envoy.extensions.filters.http.router.v3.Router
          route_config:
            name: local_route
            virtual_hosts:
            - name: backend
              domains: ["*"]
              routes:
              - match:
                  prefix: "/api/products"
                route:
                  cluster: product_service
                  timeout: 10s
                  retry_policy:
                    retry_on:
connect-failure,refused-stream,unavailable,cancelled,resource-exhausted
                    num_retries: 3
                    retry_host_predicate:
                    - name: envoy.retry_host_predicates.previous_hosts
                    host_selection_retry_max_attempts: 3
                    retriable_status_codes: [503]
              - match:
                  prefix: "/api/orders"
                route:
                  cluster: order_service
                  timeout: 15s
```

Service Mesh and API Gateway Security Integration:

1. North-South vs. East-West Traffic:
    - API Gateway: Handles external (north-south) traffic
    - Service Mesh: Manages internal (east-west) communication
    - Both: Apply consistent security policies
2. Security Responsibility Split:

    - API Gateway:
        - External authentication/authorization
        - Rate limiting
        - Input validation
        - Edge-level threat protection
    - Service Mesh:
        - Service identity
        - Mutual TLS
        - Service-to-service authorization
        - Traffic encryption
3. Implementation Approaches:
    - Layered approach (API Gateway → Service Mesh → Services)
    - Integrated approach (API Gateway as entry point to mesh)
    - Hybrid approach (combining aspects of both)

Advanced Threat Protection for API Gateways

Modern API gateways require sophisticated threat protection capabilities:

```
# Example Python code for advanced API Gateway WAF rules
import re
from dataclasses import dataclass
from enum import Enum
from typing import Dict, List, Optional, Pattern, Set, Tuple, Union

class RuleAction(Enum):
    ALLOW = "allow"
    BLOCK = "block"
    MONITOR = "monitor"
    CHALLENGE = "challenge"

class RuleSeverity(Enum):
    LOW = "low"
    MEDIUM = "medium"
    HIGH = "high"
    CRITICAL = "critical"

@dataclass
class WafRule:
    id: str
    name: str
    description: str
    patterns: List[Pattern]
    action: RuleAction
    severity: RuleSeverity
    targets: Set[str]  # e.g. "body", "query", "headers", "uri"

    def evaluate(self, request_data: Dict[str, str]) -> Tuple[bool,
```

```
Optional[str]]:
        """Evaluate if rule matches any part of the request."""
        for target in self.targets:
            if target not in request_data:
                continue

            target_data = request_data[target]
            for pattern in self.patterns:
                if pattern.search(target_data):
                    return True, f"Rule {self.id}: {self.name} matched {target}"

        return False, None

# Example WAF rule implementation
class ApiWaf:
    def __init__(self):
        self.rules: List[WafRule] = []
        self._initialize_rules()

    def _initialize_rules(self):
        # SQL Injection rules
        self.rules.append(WafRule(
            id="sql-001",
            name="SQL Injection - Basic",
            description="Detects basic SQL injection attempts",
            patterns=[

re.compile(r"(?i)('|\")?\s*(OR|AND)\s*('|\")?\s*\d+\s*=\s*\d+\s*--"),
                re.compile(r"(?i);\s*DROP\s+TABLE"),
                re.compile(r"(?i)UNION\s+SELECT"),
                re.compile(r"(?i)SELECT\s+.*FROM\s+information_schema"),
                re.compile(r"(?i)INSERT\s+INTO.*VALUES")
            ],
            action=RuleAction.BLOCK,
            severity=RuleSeverity.HIGH,
            targets={"body", "query", "uri"}
        ))

        # XSS rules
        self.rules.append(WafRule(
            id="xss-001",
            name="Cross-Site Scripting - Basic",
            description="Detects basic XSS attempts",
            patterns=[
                re.compile(r"(?i)<script>"),
                re.compile(r"(?i)javascript:"),
                re.compile(r"(?i)onerror="),
                re.compile(r"(?i)onload="),
                re.compile(r"(?i)eval\("),
                re.compile(r"(?i)document\.cookie")
            ],
```

```
            action=RuleAction.BLOCK,
            severity=RuleSeverity.HIGH,
            targets={"body", "query", "uri"}
        ))

        # JWT attack rules
        self.rules.append(WafRule(
            id="jwt-001",
            name="JWT None Algorithm Attack",
            description="Detects JWT None algorithm attacks",
            patterns=[
re.compile(r"eyJ[a-zA-Z0-9_-]*\.eyJ[a-zA-Z0-9_-]*\.[a-zA-Z0-9_-]{0,3}$")
            ],
            action=RuleAction.BLOCK,
            severity=RuleSeverity.CRITICAL,
            targets={"headers"}
        ))

        # GraphQL Introspection
        self.rules.append(WafRule(
            id="graphql-001",
            name="GraphQL Introspection Query",
            description="Detects GraphQL introspection queries",
            patterns=[
                re.compile(r'(?i)query\s+[^{]*{?\s*__schema')
            ],
            action=RuleAction.MONITOR,
            severity=RuleSeverity.MEDIUM,
            targets={"body"}
        ))

        # Command Injection
        self.rules.append(WafRule(
            id="cmd-001",
            name="Command Injection",
            description="Detects OS command injection attempts",
            patterns=[
re.compile(r"(?i)[;|&]?\s*(cat|ls|pwd|whoami|wget|curl|nc|bash|sh|sudo|chmod)"),
re.compile(r"(?i)[;|&]?\s*(cmd\.exe|powershell|net\s+user|systeminfo|tasklist)")
,
                re.compile(r"(?i)/etc/(passwd|shadow|hosts)")
            ],
            action=RuleAction.BLOCK,
            severity=RuleSeverity.CRITICAL,
            targets={"body", "query", "uri"}
        ))

        # API-specific attacks
```

```
        self.rules.append(WafRule(
            id="api-001",
            name="Excessive API Nesting",
            description="Detects excessively nested API fields that could lead
to DoS",
            patterns=[
                re.compile(r"[^{}]*(\{[^{}]*){5,}")
            ],
            action=RuleAction.BLOCK,
            severity=RuleSeverity.MEDIUM,
            targets={"body"}
        ))

    def evaluate_request(self, request_data: Dict[str, str]) -> List[Dict]:
        """Evaluate all rules against the request."""
        results = []

        for rule in self.rules:
            matched, message = rule.evaluate(request_data)

            if matched:
                results.append({
                    "rule_id": rule.id,
                    "rule_name": rule.name,
                    "severity": rule.severity.value,
                    "action": rule.action.value,
                    "message": message
                })

                # If any blocking rule matches, return immediately
                if rule.action == RuleAction.BLOCK:
                    return results

        return results

# Example middleware usage
def waf_middleware(request):
    """Example middleware for WAF implementation."""
    waf = ApiWaf()

    # Construct request data for WAF evaluation
    request_data = {
        "uri": request.path,
        "query": request.query_string.decode('utf-8'),
        "body": request.get_data(as_text=True),
        "headers": '\n'.join([f"{k}: {v}" for k, v in request.headers.items()])
    }

    # Evaluate request against WAF rules
    results = waf.evaluate_request(request_data)
```

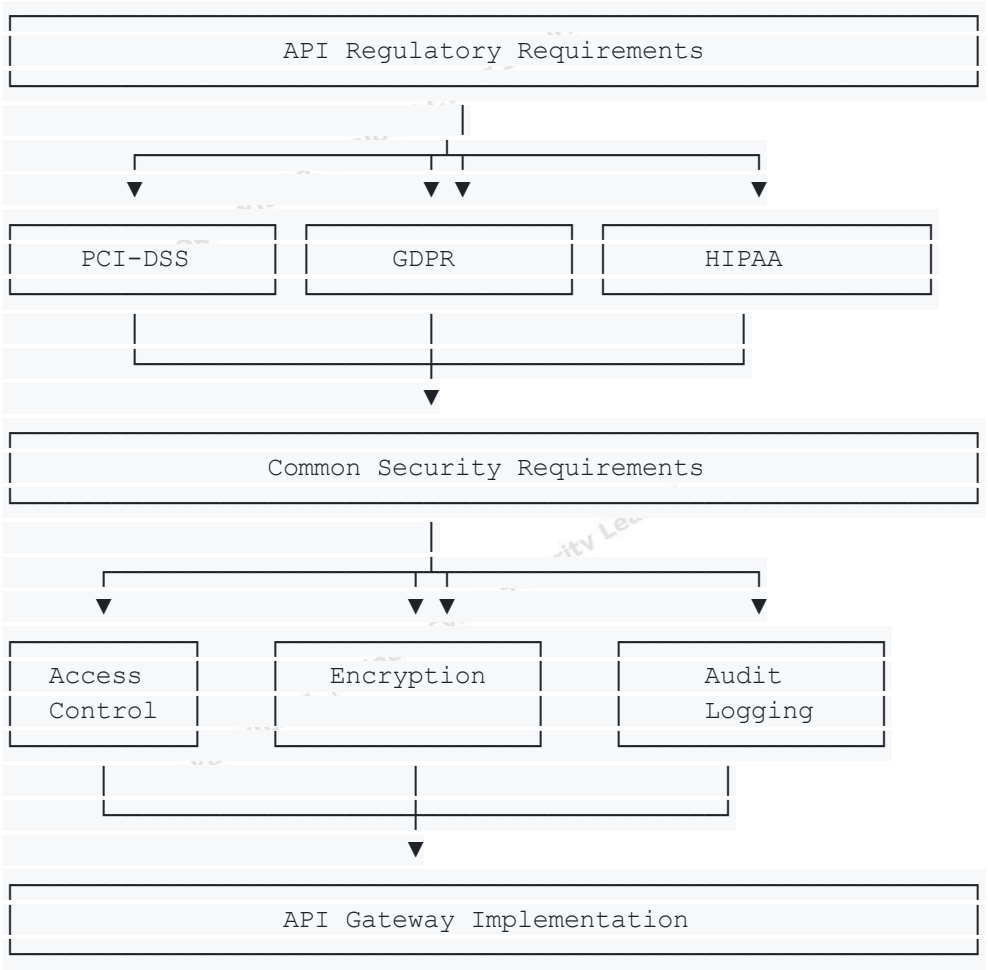
```
    # Check if any blocking rule matched
    for result in results:
        if result["action"] == "block":
            return {
                "error": "Request blocked by security rules",
                "rule": result["rule_id"],
                "detail": result["message"]
            }, 403

    # Log any monitored rule matches
    for result in results:
        if result["action"] == "monitor":
            log_security_event(request, result)

    # Continue with request processing
    return None
```

Advanced API Threat Protection Strategies:

1. Behavioral Analysis:
    - Establish baseline API usage patterns
    - Detect anomalies in request patterns or content
    - Identify unusual API sequence calls
    - Monitor for changes in client behavior
2. Business Logic Attack Protection:
    - Implement API sequence enforcement
    - Detect enumeration and resource harvesting
    - Protect against parameter tampering
    - Identify business process abuse
3. API-Specific Protections:
    - GraphQL query depth and complexity limitations
    - REST resource exhaustion protection
    - SOAP XML entity attack prevention
    - API parameter pollution detection

Compliance and API Gateway Security

Regulatory Requirements for API Security



Key Compliance Requirements for API Gateways:

1. PCI DSS Requirements:
    - Implement strong access control measures (Requirement 7, 8)
    - Encrypt transmission of cardholder data (Requirement 4)
    - Regularly test security systems and processes (Requirement 11)
    - Maintain a vulnerability management program (Requirement 6)
    - Track and monitor all access to network resources (Requirement 10)
2. GDPR Considerations:
    - Implement data protection by design and default
    - Ensure lawful basis for processing through proper validation
    - Support data subject rights (access, deletion, portability)
    - Maintain records of processing activities
    - Enable data minimization through API controls
3. HIPAA Requirements:
    - Implement technical safeguards for PHI
    - Ensure proper authentication and authorization
    - Maintain comprehensive audit trails

- Implement transmission security for PHI
- Support emergency access procedures

Audit Trail Implementation

```
-- Example database schema for API Gateway audit logging
CREATE TABLE api_audit_log (
    id BIGSERIAL PRIMARY KEY,
    request_id VARCHAR(36) NOT NULL,
    correlation_id VARCHAR(36),
    timestamp TIMESTAMP WITH TIME ZONE NOT NULL,
    client_ip INET NOT NULL,
    api_key_id VARCHAR(64),
    user_id VARCHAR(64),
    http_method VARCHAR(10) NOT NULL,
    resource_path TEXT NOT NULL,
    query_string TEXT,
    request_headers JSONB,
    request_body TEXT,
    status_code INT NOT NULL,
    response_time INT NOT NULL, -- milliseconds
    error_message TEXT,
    auth_decision VARCHAR(20),
    threat_detected BOOLEAN DEFAULT FALSE,
    threat_details JSONB,
    geo_location JSONB,
    user_agent TEXT,
    created_at TIMESTAMP WITH TIME ZONE DEFAULT NOW()
);

-- Create indexes for efficient querying
CREATE INDEX idx_audit_timestamp ON api_audit_log(timestamp);
CREATE INDEX idx_audit_user_id ON api_audit_log(user_id);
CREATE INDEX idx_audit_resource_path ON api_audit_log(resource_path);
CREATE INDEX idx_audit_status_code ON api_audit_log(status_code);
CREATE INDEX idx_audit_client_ip ON api_audit_log(client_ip);
CREATE INDEX idx_audit_threat_detected ON api_audit_log(threat_detected);

-- Example query for security analysis
-- Find suspicious patterns - multiple 401/403 errors followed by successful
access
WITH suspicious_patterns AS (
    SELECT
        client_ip,
        user_id,
        resource_path,
        COUNT(*) FILTER (WHERE status_code IN (401, 403)) AS failed_attempts,
        COUNT(*) FILTER (WHERE status_code = 200) AS successful_attempts,
        MIN(timestamp) FILTER (WHERE status_code = 200) AS first_success_time,
        MAX(timestamp) FILTER (WHERE status_code IN (401, 403)) AS
```

```
last_failure_time
    FROM api_audit_log
    WHERE timestamp > NOW() - INTERVAL '1 hour'
    GROUP BY client_ip, user_id, resource_path
    HAVING
        COUNT(*) FILTER (WHERE status_code IN (401, 403)) > 5
        AND COUNT(*) FILTER (WHERE status_code = 200) > 0
)
SELECT
    sp.*,
    EXTRACT(EPOCH FROM (first_success_time - last_failure_time)) AS
seconds_between_failure_and_success
FROM suspicious_patterns sp
WHERE first_success_time > last_failure_time
AND EXTRACT(EPOCH FROM (first_success_time - last_failure_time)) < 300; -- 5
minutes
```

Future Trends in API Gateway Security

Emerging API Security Technologies

The API security landscape continues to evolve with several emerging trends:

1. AI-Powered API Security:
    - Machine learning for anomaly detection
    - Automated threat response based on behavioral analysis
    - Predictive API security based on threat intelligence
    - Self-healing API configurations
2. Zero Trust API Networks:
    - Identity-aware proxies
    - Continuous authentication and authorization
    - Context-based access policies
    - Per-request security evaluation
3. API Infrastructure as Code Security:
    - Security policy as code
    - Automated security testing in CI/CD
    - Infrastructure compliance validation
    - Runtime security enforcement
4. Decentralized Identity for APIs:
    - Self-sovereign identity integration
    - Decentralized PKI
    - Verifiable credentials for API access
    - DID (Decentralized Identifier) authentication

Conclusion

API gateways represent a critical security boundary in modern architectures, functioning as the primary control point for all API traffic. Implementing a robust security strategy for API gateways requires a multi-layered approach that addresses authentication, authorization, rate limiting, input validation, monitoring, and threat protection.

By following the best practices outlined in this guide, organizations can significantly enhance their API security posture, protecting both their systems and data from increasingly sophisticated attacks targeting API infrastructures. As API-driven architectures continue to evolve, maintaining strong API gateway security will remain a critical priority for security teams.

Remember that API gateway security is not a one-time implementation but an ongoing process that requires continuous assessment, improvement, and adaptation to emerging threats and technologies. By establishing a comprehensive security program around your API gateways, you can ensure they remain effective guardians of your digital assets.

Frequently Asked Questions

How does API gateway security differ from traditional network security?

API gateway security differs from traditional network security in several key ways:

1. Protocol Focus:
    - Traditional network security focuses on network protocols (TCP/IP, UDP) and packet filtering
    - API gateway security focuses on application-layer protocols (HTTP/HTTPS) and API-specific threats
2. Authentication Approaches:
    - Traditional: Network-based authentication (VPNs, firewalls)
    - API gateways: Identity-based authentication (OAuth, JWT, API keys)
3. Traffic Analysis:
    - Traditional: Packet inspection, network flow analysis
    - API gateways: API call patterns, payload inspection, business logic validation
4. Security Boundaries:
    - Traditional: Network perimeters, segments
    - API gateways: Service boundaries, fine-grained API controls

5. Threat Models:
    - Traditional: Network-based attacks (DDoS, port scanning)
    - API gateways: API-specific attacks (injection, broken authentication, excessive data exposure)

What are the security considerations for transitioning from monolithic to microservices architecture with API gateways?

Transitioning from monolith to microservices introduces several security challenges that API gateways can help address:

1. Authentication Centralization:
    - Monolith: Single authentication system
    - Microservices: Multiple services require authentication
    - Solution: Centralize authentication at the API gateway
2. Authorization Complexity:
    - Monolith: Unified authorization model
    - Microservices: Distributed authorization decisions
    - Solution: Implement policy-based access control at the gateway level
3. Increased Attack Surface:
    - Monolith: Single deployment unit
    - Microservices: Multiple services and communication paths
    - Solution: Apply consistent security controls via the gateway
4. Service-to-Service Communication:
    - Monolith: Internal function calls
    - Microservices: Network calls between services
    - Solution: Implement service mesh with mutual TLS alongside API gateway
5. Consistent Security Policies:
    - Monolith: Single codebase for security controls
    - Microservices: Distributed implementation of security
    - Solution: Define and enforce security policies at the gateway

How can organizations implement effective API security testing for gateways?

A comprehensive API gateway security testing program should include:

1. Automated Security Scanning:
    - API-specific DAST tools (OWASP ZAP, Burp Suite)
    - Custom security scanners for API gateway configurations
    - Continuous scanning integrated with CI/CD

2. Manual Penetration Testing:
    - Targeted testing of authentication mechanisms
    - Authorization bypass attempts
    - Business logic testing
    - API abuse scenarios
3. Configuration Analysis:
    - Gateway configuration review
    - Security policy validation
    - TLS configuration testing
    - JWT/OAuth implementation review
4. Performance and Resilience Testing:
    - Rate limiting effectiveness
    - DDoS resilience
    - Failover and circuit breaking
    - Resource exhaustion testing
5. Runtime Security Validation:
    - Dynamic policy enforcement testing
    - Real-time monitoring validation
    - Anomaly detection testing
    - Incident response simulation

What are the key considerations for multi-cloud API gateway security?

Securing API gateways in multi-cloud environments requires addressing several unique challenges:

1. Consistent Security Controls:
    - Implement standard security models across cloud providers
    - Use abstraction layers for cloud-agnostic security policies
    - Maintain consistent authentication mechanisms
2. Identity Federation:
    - Implement centralized identity management
    - Federate authentication across clouds
    - Support cross-cloud trust relationships
3. Security Monitoring:
    - Aggregate logs from all environments
    - Implement cross-cloud security monitoring
    - Enable comprehensive visibility across deployments
4. Compliance Challenges:
    - Address varied compliance requirements by region/cloud
    - Implement data residency controls
    - Maintain audit trails across environments
    - Enforce consistent security baselines

5. Network Security Considerations:
    - Secure inter-cloud communications
    - Implement private connectivity options
    - Apply consistent encryption standards
    - Manage multi-cloud network policies
6. Automation and Infrastructure as Code:
    - Use infrastructure as code for consistent deployment
    - Implement security policy as code
    - Automate security testing across environments
    - Enable automated remediation

What security metrics should be tracked for API gateways?

Measuring API gateway security effectiveness requires tracking several key metrics:

1. Security Incident Metrics:
    - Number of security incidents by type
    - Mean time to detect (MTTD)
    - Mean time to respond (MTTR)
    - Incident impact severity
2. Authentication and Authorization Metrics:
    - Authentication failure rate
    - Token revocation frequency
    - Authorization denial rate
    - Invalid JWT token attempts
3. Rate Limiting and Abuse Metrics:
    - Rate limit hit frequency
    - API abuse attempts detected
    - Blocked request percentage
    - Traffic anomaly detections
4. Vulnerability Management:
    - Open vulnerabilities by severity
    - Mean time to remediate vulnerabilities
    - Vulnerability recurrence rate
    - Security debt metrics
5. Compliance Metrics:
    - Policy compliance percentage
    - Security control coverage
    - Compliance gaps identified
    - Failed security tests percentage

Summary

API gateway security represents a critical component of modern application security architecture. By implementing comprehensive security controls at the API gateway layer, organizations can establish a robust security perimeter that protects backend services from a wide range of threats.

Key takeaways from this guide include:

1. Defense-in-Depth: Implement multiple security layers including authentication, authorization, encryption, rate limiting, and threat protection.
2. Standardized Controls: Leverage industry standards like OAuth 2.0, OIDC, and mTLS for authentication and authorization.
3. Proactive Protection: Use advanced rate limiting, input validation, and API-specific threat protection to prevent attacks.
4. Comprehensive Monitoring: Implement detailed logging, monitoring, and analytics to detect and respond to security incidents.
5. Zero Trust Architecture: Apply zero trust principles to API security by verifying every request, regardless of source.
6. Continuous Security: Treat API security as an ongoing process with regular testing, updating, and improvement.

As API architectures continue to evolve, maintaining strong API gateway security practices will remain essential for protecting organizations' digital assets and ensuring the integrity, confidentiality, and availability of their services.