Contents

1 SEC-010: Missing Rate Limiting

1.1 Severity

Medium

1.2 OWASP Reference

A04:2021 - Insecure Design

1.3 CWE References

- CWE-307: Improper Restriction of Rendered UI Layers or Frames
- CWE-770: Allocation of Resources Without Limits or Throttling

1.4 Description

Missing Rate Limiting vulnerabilities occur when an application fails to implement request throttling or rate limiting mechanisms, allowing attackers to flood the server with requests. In MCP servers, this enables brute-force attacks, credential stuffing, API abuse, denial-of-service attacks, or resource exhaustion. Without rate limiting, attackers can attempt unlimited authentication tries, spam operations, or exhaust server resources.

1.5 Compliance Mapping

1.5.1 SOC2

- **CC6.1**: Access Control - Rate limiting prevents unauthorized access attempts.
- **CC6.8**: System Operations - The organization prevents resource exhaustion through rate limiting.
- **CC7.2**: System Monitoring - The organization monitors and detects abnormal traffic patterns.

1.5.2 HIPAA

- **§164.312(b)**: Audit Controls - Implement monitoring for suspicious access patterns and brute-force attempts.
- **§164.312(a)(1)**: Access Control - Rate limiting protects against brute-force attacks on ePHI access.

1.5.3 PCI DSS

- **6.5.10**: Brute-force attacks are prevented through rate limiting and account lockout
- **11.3**: Regular security testing includes testing for brute-force resistance
- **10.2**: All access attempts including failed attempts must be monitored

1.6 Vulnerable Code Example

```
// □ INSECURE: No rate limiting on login
const express = require('express');
const app = express();

app.use(express.json());

// Vulnerable endpoint: unlimited login attempts
app.post('/api/login', async (req, res) => {
  const { username, password } = req.body;

  // No rate limiting - attacker can try unlimited credentials
  const user = await authenticateUser(username, password);

  if (user) {
    res.json({ success: true, token: generateToken(user) });
  } else {
    res.status(401).json({ error: 'Invalid credentials' });
  }
});

// Vulnerable endpoint: no limits on API calls
app.get('/api/data', (req, res) => {
  // Attacker can make unlimited requests
  const data = fetchData();
  res.json(data);
});

app.listen(3000);
```

1.7 Secure Implementation

```
// ✅ SECURE: Rate limiting with express-rate-limit
const express = require('express');
const rateLimit = require('express-rate-limit');
const RedisStore = require('rate-limit-redis');
const redis = require('redis');
const app = express();

app.use(express.json());

// Create Redis client for distributed rate limiting
const client = redis.createClient({
  host: process.env.REDIS_HOST || 'localhost',
  port: process.env.REDIS_PORT || 6379
});

// Strict rate limiter for login (5 attempts per 15 minutes)
const loginLimiter = rateLimit({
  store: new RedisStore({
    client: client,
    prefix: 'login-limit:'
  }),
  windowMs: 15 * 60 * 1000, // 15 minutes
  max: 5, // 5 requests per windowMs
  message: 'Too many login attempts, please try again later',
  standardHeaders: true,
  legacyHeaders: false,
  skip: (req) => {
    // Skip rate limiting for admin IPs
    const adminIPs = process.env.ADMIN_IPS?.split(',') || [];
    return adminIPs.includes(req.ip);
  }
});

// General API rate limiter (100 requests per 15 minutes)
const generalLimiter = rateLimit({
  store: new RedisStore({
    client: client,
    prefix: 'general-limit:'
  }),
  windowMs: 15 * 60 * 1000,
  max: 100,
  standardHeaders: true,
  legacyHeaders: false
});

// Secure endpoint: login with rate limiting
```

```
app.post('/api/login', loginLimiter, async (req, res) => {
  const { username, password } = req.body;

  try {
    const user = await authenticateUser(username, password);

    if (user) {
      // On successful login, optionally reset the rate limit
      res.json({ success: true, token: generateToken(user) });
    } else {
      res.status(401).json({ error: 'Invalid credentials' });
    }
  } catch (error) {
    res.status(500).json({ error: 'Authentication failed' });
  }
});

// Secure endpoint: API data with general rate limiting
app.get('/api/data', generalLimiter, (req, res) => {
  const data = fetchData();
  res.json(data);
});

app.listen(3000);
```

1.8 Remediation Steps

1. **Implement Rate Limiting on All Endpoints**: Apply strict rate limiting to authentication endpoints (5-10 requests per 15 minutes). Use moderate limits for general API endpoints (100-1000 per hour). Use distributed rate limiting (Redis/Memcached) for multi-server deployments. Track rate limits by IP address, user ID, and API key.

2. **Use Exponential Backoff for Retries**: Implement progressive delays after failed attempts. Lock accounts temporarily after multiple failed login attempts. Provide clear feedback to users about rate limit status. Implement gradual rate limit increase for legitimate users over time.

3. **Monitor and Alert on Rate Limit Violations**: Log all rate limit violations for security analysis. Alert security teams on suspicious patterns (multiple IPs hitting limits, bot-like behavior). Implement CAPTCHA after repeated failures. Use anomaly detection to identify coordinated attacks.

4. **Implement Tiered Rate Limiting**: Different limits for authenticated vs. unauthenticated users. Higher limits for premium/trusted users. Lower limits for new/untrusted users. Whitelist critical IPs (monitoring systems, internal services).

1.9 References

- OWASP Rate Limiting Cheat Sheet
- CWE-770: Allocation of Resources Without Limits
- express-rate-limit Documentation

Last Updated: January 2026
Status: Published
Language: English