Contents

1 SEC-011: Regular Expression Denial of Service (ReDoS)

1.1 Severity

Medium □

1.2 OWASP Reference

A04:2021 - Insecure Design

1.3 CWE References

- CWE-1333: Inefficient Regular Expression Complexity

1.4 Description

Regular Expression Denial of Service (ReDoS) vulnerabilities occur when an application uses inefficient regular expressions that can cause catastrophic backtracking when processing malicious input. In MCP servers, attackers can provide specially crafted strings that cause regex matching to consume excessive CPU, leading to server hang, performance degradation, or denial of service. This is particularly dangerous when regex patterns are applied to user-controlled input without safeguards.

1.5 Compliance Mapping

1.5.1 SOC2

- **CC6.6**: System Operations - The organization prevents denial-of-service attacks through efficient input validation.
- **CC6.8**: System Operations - Resource consumption is monitored to detect anomalies.
- **CC7.2**: System Monitoring - The organization monitors CPU usage and detects performance anomalies.

1.5.2 HIPAA

- **§164.312(a)(1)**: Access Control - Implement secure input validation to prevent DoS attacks on ePHI systems.
- **§164.312(b)**: Audit Controls - Monitor for unusual performance patterns indicating attacks.

1.5.3 PCI DSS

- **6.5.1**: Injection flaws including ReDoS are prevented through secure coding
- **6.2.4**: All custom code is developed with efficient input validation
- **11.3**: Security testing includes testing for ReDoS vulnerabilities

1.6 Vulnerable Code Example

```
// □ INSECURE: Vulnerable regex patterns
const express = require('express');
const app = express();

app.use(express.json());

// Vulnerable endpoint: email validation with catastrophic backtracking
app.post('/api/validate-email', (req, res) => {
  const { email } = req.body;

  // DANGER: This regex is vulnerable to ReDoS
  // Pattern: (a+)+$ causes exponential backtracking
  const emailRegex = /^(([a-z]|[a-z]{2})+)+@([a-z]+\.)+[a-z]{2,}$/i;

  // Attacker input: 'aaaaaaaaaaaaaaaaaaaaaaaaaaaaab' causes CPU spike
  if (emailRegex.test(email)) {
    res.json({ valid: true });
  } else {
    res.json({ valid: false });
  }
});

// Vulnerable endpoint: password validation
app.post('/api/validate-password', (req, res) => {
  const { password } = req.body;

  // DANGER: (w+)* allows ReDoS attack
  const passwordRegex = /^(\w+)*$/;

  // Attacker input: long string of 'w' characters causes hang
  if (passwordRegex.test(password)) {
    res.json({ valid: true });
  } else {
    res.json({ valid: false });
  }
});

app.listen(3000);
```

1.7 Secure Implementation

```
// □ SECURE: Safe regex patterns and input validation
const express = require('express');
const { check, validationResult } = require('express-validator');
const app = express();

app.use(express.json());

// Safe helper function for email validation
function isValidEmail(email) {
  // Limit input length to prevent even safe regex from processing huge strings
  if (!email || email.length > 254) {
    return false;
  }

  // Use a safe email regex pattern (no catastrophic backtracking)
  // Or better yet, use a library like validator.js
  const emailRegex = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;

  return emailRegex.test(email);
}

// Safe helper function for password validation
function isValidPassword(password) {
  if (!password || password.length < 8 || password.length > 128) {
    return false;
  }

  // Check specific criteria instead of complex regex
  const hasUpperCase = /[A-Z]/.test(password);
  const hasLowerCase = /[a-z]/.test(password);
  const hasNumber = /[0-9]/.test(password);
  const hasSpecialChar = /[!@#$%^&*]/.test(password);

  return hasUpperCase && hasLowerCase && hasNumber && hasSpecialChar;
}

// Secure endpoint: email validation with timeout
app.post('/api/validate-email', [
  check('email')
```

```
    .trim()
    .notEmpty().withMessage('Email is required')
    .isLength({ max: 254 }).withMessage('Email too long')
    .custom((value) => {
      if (!isValidEmail(value)) {
        throw new Error('Invalid email format');
      }
      return true;
    })
], (req, res) => {
  const errors = validationResult(req);

  if (!errors.isEmpty()) {
    return res.status(400).json({ errors: errors.array() });
  }

  const { email } = req.body;
  res.json({ valid: true, email });
});

// Secure endpoint: password validation with input limits
app.post('/api/validate-password', [
  check('password')
    .isLength({ min: 8, max: 128 })
    .withMessage('Password must be 8-128 characters')
    .custom((value) => {
      if (!isValidPassword(value)) {
        throw new Error('Password must contain uppercase, lowercase, number, and sp
      }
      return true;
    })
], (req, res) => {
  const errors = validationResult(req);

  if (!errors.isEmpty()) {
    return res.status(400).json({ errors: errors.array() });
  }

  res.json({ valid: true });
});

// Secure endpoint: URL validation with safe regex
app.post('/api/validate-url', [
  check('url')
    .trim()
    .isLength({ max: 2048 })
    .withMessage('URL too long')
    .isURL()
    .withMessage('Invalid URL format')
], (req, res) => {
  const errors = validationResult(req);

  if (!errors.isEmpty()) {
    return res.status(400).json({ errors: errors.array() });
  }

  const { url } = req.body;
  res.json({ valid: true, url });
});

// Secure endpoint: IP address validation
app.post('/api/validate-ip', [
  check('ip')
    .trim()
    .isIP()
    .withMessage('Invalid IP address')
], (req, res) => {
  const errors = validationResult(req);

  if (!errors.isEmpty()) {
    return res.status(400).json({ errors: errors.array() });
  }

  const { ip } = req.body;
  res.json({ valid: true, ip });
});

app.listen(3000);
```

1.8 Remediation Steps

1. **Avoid Complex Regex Patterns with Backtracking**: Use safe regex patterns without nested quantifiers (e.g., avoid (a+)+, (a)). Prefer linear-time regex patterns or use library functions (validator.js, is.js) instead of custom regex. Test regex patterns with ReDoS detection tools. Limit input length before regex processing to reduce impact.

2. **Use Input Validation Libraries Instead of Custom Regex**: Use libraries like `express-validator`, `joi`, or `yup` that have safe validation patterns. These libraries have been thoroughly tested for ReDoS vulnerabilities. Validate input format, length, and whitelist characters when possible rather than using complex patterns.

3. **Implement Timeout Mechanisms for Regex Operations**: Set execution timeouts on regex matching (Node.js doesn't have built-in regex timeouts, but external tools can help). Monitor regex execution time and alert if patterns exceed

expected duration. Use alternative validation methods (character whitelisting, string methods) for simple validations.

4. **Test and Monitor for ReDoS**: Use tools like OWASP ReDoS-Scanner or regex101.com to test patterns for vulnerabilities. Include ReDoS testing in security testing procedures. Monitor CPU usage patterns for spikes during input validation. Implement rate limiting on validation endpoints to mitigate attack impact.

1.9 References

- OWASP Regular Expression Denial of Service
- CWE-1333: Inefficient Regular Expression Complexity
- ReDoS Detection and Prevention Guide
- Safe Regex Testing Tool
- express-validator Documentation

Last Updated: January 2026
Status: Published
Language: English