Contents

1 SEC-004: Server-Side Request Forgery (SSRF)

1.1 Severity

High

1.2 OWASP Reference

A10:2021 - Server-Side Request Forgery (SSRF)

1.3 CWE References

- CWE-918: Server-Side Request Forgery (SSRF)

1.4 Description

Server-Side Request Forgery (SSRF) vulnerabilities occur when an application makes HTTP requests to URLs supplied by users without proper validation and sanitization. In MCP servers, this allows attackers to make the server perform unwanted requests to internal services, cloud metadata endpoints, or external systems, potentially leading to unauthorized access, data disclosure, cloud infrastructure compromise, or lateral movement within internal networks. This is particularly dangerous in cloud environments where metadata services are accessible via HTTP.

1.5 Compliance Mapping

1.5.1 SOC2

- **CC6.1**: Logical and Physical Access Controls - The organization restricts access to internal resources and services.
- **CC6.8**: System Operations - The organization prevents unauthorized system access through secure network communications.
- **CC7.2**: System Monitoring - The organization monitors network traffic and detects suspicious request patterns.

1.5.2 HIPAA

- **§164.312(a)(1)**: Access Control - Implement technical policies to prevent unauthorized access to internal systems containing ePHI.
- **§164.312(e)(1)**: Transmission Security - Implement technical security measures to protect ePHI from interception during transmission.
- **§164.312(c)(1)**: Integrity Controls - Protect internal ePHI systems from unauthorized access via SSRF exploitation.

1.5.3 PCI DSS

- **6.5.10**: Broken authentication via SSRF is prevented through secure coding
- **6.2.4**: Custom software is developed securely with proper request validation
- **1.3.1**: Prohibit direct public access to internal network resources
- **11.3.1**: SSRF vulnerabilities are identified through vulnerability scanning

1.6 Vulnerable Code Example

```
// □ INSECURE: Making requests to user-supplied URLs without validation
const express = require('express');
const axios = require('axios');
const app = express();

app.use(express.json());

// Vulnerable endpoint: image proxy
app.post('/api/proxy-image', async (req, res) => {
  const { imageUrl } = req.body;

  // DANGER: No validation of the URL
  // Attacker input: "http://169.254.169.254/latest/meta-data/iam/security-credent
  // This would leak AWS credentials from the metadata service
  try {
    const response = await axios.get(imageUrl, { timeout: 5000 });
    res.contentType(response.headers['content-type']);
    res.send(response.data);
  } catch (error) {
    res.status(500).json({ error: 'Failed to fetch image' });
  }
});

// Vulnerable endpoint: URL preview/fetch
app.post('/api/fetch-content', async (req, res) => {
  const { url } = req.body;

  // DANGER: Direct request to user-supplied URL
  // Attacker can probe internal services: http://localhost:8080/admin
  // Or access cloud metadata: http://169.254.169.254/
```

```
  try {
    const response = await axios.get(url, { timeout: 5000 });
    res.json({ content: response.data.substring(0, 1000) });
  } catch (error) {
    res.status(500).json({ error: error.message });
  }
});

// Vulnerable endpoint: webhook delivery
app.post('/api/send-webhook', async (req, res) => {
  const { webhookUrl, payload } = req.body;

  // DANGER: No validation of webhook URL
  // Attacker can make requests to internal services with arbitrary payloads
  try {
    await axios.post(webhookUrl, payload, { timeout: 5000 });
    res.json({ success: true });
  } catch (error) {
    res.status(500).json({ error: error.message });
  }
});

app.listen(3000);
```

1.7 Secure Implementation

```
// ✅ SECURE: Strict URL validation and internal resource protection
const express = require('express');
const axios = require('axios');
const { URL } = require('url');
const dns = require('dns').promises;
const ipaddr = require('ipaddr.js');
const app = express();

app.use(express.json());

// Blocked IP ranges (internal and special-use addresses)
const BLOCKED_IP_RANGES = [
  '127.0.0.0/8',      // localhost
  '10.0.0.0/8',       // private
  '172.16.0.0/12',    // private
  '192.168.0.0/16',   // private
  '169.254.0.0/16',   // link-local (AWS metadata service!)
  '224.0.0.0/4',      // multicast
  '255.255.255.255/32' // broadcast
];
```

```
// Whitelisted domains for specific use cases
const WHITELISTED_DOMAINS = [
  'cdn.example.com',
  'api.trusted-partner.com',
  'media.example.com'
];

// Helper function to check if IP is in blocked range
function isBlockedIp(ip) {
  try {
    const addr = ipaddr.process(ip);
    return BLOCKED_IP_RANGES.some(range => {
      const [rangeIp, prefixLength] = range.split('/');
      return addr.match(ipaddr.process(rangeIp), parseInt(prefixLength));
    });
  } catch (error) {
    return false;
  }
}

// Helper function to validate URL
async function validateUrl(urlString, whitelist = false) {
  try {
    const parsedUrl = new URL(urlString);

    // Only allow HTTP and HTTPS
    if (!['http:', 'https:'].includes(parsedUrl.protocol)) {
      throw new Error('Only HTTP(S) protocols allowed');
    }

    // If whitelist mode, check against whitelist
    if (whitelist && !WHITELISTED_DOMAINS.includes(parsedUrl.hostname)) {
      throw new Error('Domain not in whitelist');
    }

    // Check for suspicious patterns
    if (parsedUrl.hostname.includes('..') || parsedUrl.hostname.includes('..')) {
      throw new Error('Invalid hostname format');
    }

    // DNS resolution with timeout
    const addresses = await Promise.race([
      dns.resolve4(parsedUrl.hostname),
      new Promise((_, reject) =>
        setTimeout(() => reject(new Error('DNS timeout')), 5000)
      )
    ]);
```

```
    // Verify resolved IPs are not blocked
    for (const ip of addresses) {
      if (isBlockedIp(ip)) {
        throw new Error(`Resolved IP ${ip} is in blocked range`);
      }
    }

    // Verify URL length to prevent extremely long URLs
    if (urlString.length > 2048) {
      throw new Error('URL too long');
    }

    return parsedUrl;

  } catch (error) {
    throw new Error(`Invalid URL: ${error.message}`);
  }
}

// Secure endpoint: image proxy with strict validation
app.post('/api/proxy-image', async (req, res) => {
  const { imageUrl } = req.body;

  if (!imageUrl || typeof imageUrl !== 'string') {
    return res.status(400).json({ error: 'Invalid imageUrl' });
  }

  try {
    // Validate URL with whitelist for trusted CDNs
    const parsedUrl = await validateUrl(imageUrl, true);

    // Make request with strict timeout and size limits
    const response = await axios.get(parsedUrl.toString(), {
      timeout: 5000,
      maxContentLength: 10 * 1024 * 1024, // 10MB max
      maxRedirects: 1, // Limit redirects
      validateStatus: (status) => status >= 200 && status < 300 // Only 2xx
    });

    // Validate content type
    const contentType = response.headers['content-type'];
    if (!contentType || !contentType.includes('image/')) {
      return res.status(400).json({ error: 'Invalid content type' });
    }

    res.contentType(contentType);
    res.send(response.data);
```

```
  } catch (error) {
    console.error('Proxy error:', error);
    res.status(400).json({ error: 'Cannot fetch image' });
  }
});

// Secure endpoint: fetch content with restricted access
app.post('/api/fetch-content', async (req, res) => {
  const { url } = req.body;

  if (!url || typeof url !== 'string') {
    return res.status(400).json({ error: 'Invalid URL' });
  }

  try {
    // Validate URL (no whitelist - more permissive but still protected)
    const parsedUrl = await validateUrl(url, false);

    // Additional validation: port restrictions
    const port = parsedUrl.port ? parseInt(parsedUrl.port) : (parsedUrl.protocol ==

    // Block common internal service ports
    const BLOCKED_PORTS = [
      22,    // SSH
      3306,  // MySQL
      5432,  // PostgreSQL
      6379,  // Redis
      27017, // MongoDB
      9200,  // Elasticsearch
      8080,  // Common internal service
      8443   // Common internal service
    ];

    if (BLOCKED_PORTS.includes(port)) {
      return res.status(403).json({ error: 'Access to this port is blocked' });
    }

    const response = await axios.get(parsedUrl.toString(), {
      timeout: 5000,
      maxContentLength: 1 * 1024 * 1024, // 1MB max for content
      maxRedirects: 2,
      headers: {
        'User-Agent': 'MCP-SafeFetcher/1.0' // Identify as MCP service
      }
    });

    // Sanitize response before returning
    const contentLength = response.data.length;
```

```
    res.json({
      success: true,
      contentLength,
      preview: typeof response.data === 'string'
        ? response.data.substring(0, 500)
        : 'Binary content'
    });

  } catch (error) {
    console.error('Fetch error:', error);
    res.status(400).json({ error: 'Cannot fetch content' });
  }
});

// Secure endpoint: webhook delivery with validated destination
app.post('/api/send-webhook', async (req, res) => {
  const { webhookUrl, payload } = req.body;

  if (!webhookUrl || typeof webhookUrl !== 'string') {
    return res.status(400).json({ error: 'Invalid webhookUrl' });
  }

  if (!payload || typeof payload !== 'object') {
    return res.status(400).json({ error: 'Invalid payload' });
  }

  try {
    // Validate webhook URL
    const parsedUrl = await validateUrl(webhookUrl, false);

    // Verify it's a valid webhook destination
    const hostname = parsedUrl.hostname;
    if (hostname === 'localhost' || hostname === '127.0.0.1') {
      return res.status(403).json({ error: 'Cannot send webhook to localhost' });
    }

    // Make webhook request with strict parameters
    const response = await axios.post(
      parsedUrl.toString(),
      payload,
      {
        timeout: 10000,
        maxRedirects: 0, // No redirects for webhooks
        headers: {
          'Content-Type': 'application/json',
          'User-Agent': 'MCP-Webhook/1.0'
        }
      }
```

```
    );

    res.json({
      success: true,
      statusCode: response.status
    });

  } catch (error) {
    console.error('Webhook error:', error);
    res.status(400).json({ error: 'Webhook delivery failed' });
  }
});

app.listen(3000);
```

1.8 Remediation Steps

1. **Implement URL Validation and Filtering**: Create a robust URL validation function that checks protocol (only HTTP/HTTPS), performs DNS resolution with timeout, validates resolved IPs against blocked ranges (127.0.0.0/8, 10.0.0.0/8, 172.16.0.0/12, 192.168.0.0/16, 169.254.0.0/16, etc.), and enforces reasonable length limits. Use whitelisting for sensitive operations (image proxying) and greylist/blacklist for general operations.
2. **Block Internal IP Ranges and Metadata Services**: Explicitly block private IP ranges, loopback addresses, and cloud provider metadata endpoints (AWS: 169.254.169.254, GCP: metadata.google.internal, Azure: 169.254.169.254). Implement DNS rebinding protection by re-resolving hostnames before making requests. Restrict outbound traffic to specific ports (block 22, 3306, 5432, 6379, 27017, 9200, etc.).
3. **Implement Rate Limiting and Request Isolation**: Limit the number of requests per user/IP per time period. Use separate network namespaces or containers for making outbound requests. Implement request timeouts (5-10 seconds) and strict size limits (1-10MB depending on use case). Log all outbound requests for audit and investigation.
4. **Apply Defense in Depth with Monitoring**: Use network policies to restrict outbound access to specific destinations. Implement Web Application Firewalls (WAF) with SSRF detection rules. Monitor for unusual DNS patterns, multiple failed connection attempts, or requests to internal ranges. Conduct regular security reviews of code handling user-supplied URLs. Use tools like Burp Suite's SSRF hunting tools for testing.

1.9 References

- OWASP Server-Side Request Forgery (SSRF)
- CWE-918: Server-Side Request Forgery
- OWASP SSRF Prevention Cheat Sheet

- PortSwigger: Server-Side Request Forgery
- AWS Security: Mitigating SSRF Attacks

Last Updated: January 2026
Status: Published
Language: English