Contents

1 SEC-008: Data Leakage in Responses

1.1 Severity

Medium

1.2 OWASP Reference

A01:2021 - Broken Access Control

1.3 CWE References

- CWE-200: Exposure of Sensitive Information to an Unauthorized Actor
- CWE-209: Information Exposure Through an Error Message

1.4 Description

Data Leakage in Responses occurs when an API returns more sensitive information than necessary, exposing unauthorized data to clients. This includes returning full user records when only email is needed, exposing internal system details in responses, returning timestamps that reveal system behavior, or including stack traces in error responses. In MCP servers, this can expose business logic, internal identifiers, or sensitive user information to unauthorized parties.

1.5 Compliance Mapping

1.5.1 SOC2

- **CC6.1**: Logical and Physical Access Controls - Data is classified and only necessary information is exposed.
- **CC6.2**: Access Control - Field-level access controls restrict sensitive data exposure.
- **CC7.2**: System Monitoring - Responses are monitored to detect unintended data leakage.

1.5.2 HIPAA

- **§164.312(a)(1)**: Access Control - Return only minimum necessary ePHI in API responses.
- **§164.312(c)(1)**: Integrity Controls - Verify only authorized data is returned to users.

1.5.3 PCI DSS

- **6.5.5**: Insecure direct object references are prevented by validating data exposure
- **3.2.1**: Cardholder data is not stored or returned in API responses
- **10.1**: Implement logging of all access to audit what data is returned

1.6 Vulnerable Code Example

```
// □ INSECURE: Returning unnecessary sensitive data
const express = require('express');
const app = express();

// Vulnerable endpoint: return all user fields
app.get('/api/users/:id', (req, res) => {
  const user = {
    id: 123,
    username: 'john_doe',
    email: 'john@example.com',
    password_hash: '$2b$12$...', // Exposed!
    social_security_number: '123-45-6789', // Exposed!
    credit_card: '4532-1234-5678-9012', // Exposed!
    internal_employee_id: 'EMP-12345', // Exposed!
    api_key: 'sk_live_abc123xyz', // Exposed!
    home_address: '123 Main St', // Exposed!
    phone: '555-1234' // Exposed!
  };

  res.json(user); // Returns everything!
});

// Vulnerable endpoint: error with stack trace
app.get('/api/data', (req, res) => {
  try {
    throw new Error('Database connection failed');
  } catch (error) {
    // DANGER: Stack trace reveals internal paths
    res.status(500).json({
      error: error.message,
      stack: error.stack // Exposes file paths and function names
    });
```

```
  }
});

// Vulnerable endpoint: list all users with sensitive data
app.get('/api/users', (req, res) => {
  const users = [
    { id: 1, username: 'admin', password_hash: '...', role: 'admin', ssn: '111-11-
    { id: 2, username: 'user', password_hash: '...', role: 'user', ssn: '222-22-22
  ];

  res.json(users); // All sensitive fields exposed!
});

app.listen(3000);
```

1.7 Secure Implementation

```
// ✅ SECURE: Return only necessary, non-sensitive data
const express = require('express');
const app = express();

// Define response schemas for different contexts
const userSchemas = {
  // Public profile (what other users can see)
  publicProfile: (user) => ({
    id: user.id,
    username: user.username,
    created_at: user.created_at
  }),

  // Private profile (what authenticated user can see about themselves)
  privateProfile: (user) => ({
    id: user.id,
    username: user.username,
    email: user.email,
    created_at: user.created_at,
    updated_at: user.updated_at
  }),

  // Admin view (what admins can see)
  adminView: (user) => ({
    id: user.id,
    username: user.username,
    email: user.email,
    role: user.role,
    created_at: user.created_at,
    last_login: user.last_login,
```

```
    status: user.status
  }),

  // List view (minimal data for listings)
  listView: (user) => ({
    id: user.id,
    username: user.username,
    status: user.status
  })
};

// Secure endpoint: return filtered user data
app.get('/api/users/:id', (req, res) => {
  const userId = parseInt(req.params.id);

  if (isNaN(userId)) {
    return res.status(400).json({ error: 'Invalid user ID' });
  }

  // Simulate fetching user
  const user = {
    id: userId,
    username: 'john_doe',
    email: 'john@example.com',
    password_hash: '$2b$12$...',
    ssn: '123-45-6789',
    role: 'user',
    created_at: '2024-01-01T00:00:00Z'
  };

  // Determine which schema to use based on auth context
  let responseData;

  if (req.user.id === userId) {
    // User viewing their own profile
    responseData = userSchemas.privateProfile(user);
  } else if (req.user.role === 'admin') {
    // Admin viewing another user
    responseData = userSchemas.adminView(user);
  } else {
    // Other users viewing public profile
    responseData = userSchemas.publicProfile(user);
  }

  res.json(responseData);
});

// Secure endpoint: list users with minimal data
```

```
app.get('/api/users', (req, res) => {
  const users = [
    { id: 1, username: 'admin', role: 'admin', created_at: '2024-01-01' },
    { id: 2, username: 'user', role: 'user', created_at: '2024-01-02' }
  ];

  // Return only list view data
  const filtered = users.map(u => userSchemas.listView(u));

  res.json({
    total: filtered.length,
    items: filtered
  });
});

// Secure endpoint: error handling without exposing details
app.get('/api/data', (req, res) => {
  try {
    throw new Error('Database connection failed');
  } catch (error) {
    // Log full error server-side
    console.error('Full error details:', error);

    // Return generic error to client
    res.status(500).json({
      error: 'Internal server error',
      errorId: 'ERR_DB_001' // Can be used for support tickets
    });
  }
});

// Secure endpoint: search without exposing all data
app.get('/api/search', (req, res) => {
  const { query } = req.query;

  if (!query) {
    return res.status(400).json({ error: 'Search query required' });
  }

  // Simulate search results
  const results = [
    { id: 1, username: 'john_doe', created_at: '2024-01-01' },
    { id: 2, username: 'jane_doe', created_at: '2024-01-02' }
  ];

  // Return only safe fields for search results
  const filtered = results.map(u => ({
    id: u.id,
```

```
    username: u.username,
    // Don't include timestamps that might reveal info
  }));

  res.json({
    query: query,
    results: filtered,
    count: filtered.length
  });
});

app.listen(3000);
```

1.8 Remediation Steps

1. **Define Response Schemas by Context**: Create explicit response schemas for different use cases (public view, private view, admin view, list view). Only include fields necessary for each context. Use field-level access controls to determine which data to expose. Document what data is exposed in each API response.
2. **Implement Field Filtering at Data Retrieval**: Filter sensitive fields at the database query level when possible. Use ORM projections to select only necessary columns. Never fetch unnecessary data then filter it in code. Audit database queries to ensure minimal data retrieval.
3. **Handle Errors Securely**: Never expose stack traces, file paths, or internal error details to clients. Return generic error messages to clients while logging full details server-side. Create error IDs for tracking/support without revealing system details. Sanitize all error messages before returning to clients.
4. **Monitor and Log Data Exposure**: Implement logging of all API responses (redacted sensitive data). Use monitoring to detect unusual response sizes or unexpected data fields. Regularly audit API responses for unintended data exposure. Implement rate limiting on read endpoints to prevent data harvesting.

1.9 References

- OWASP API Security - Excessive Data Exposure
- CWE-200: Exposure of Sensitive Information
- OWASP API Response Security

Last Updated: January 2026
Status: Published
Language: English