Cybersecurity Vulnerability Assessment Report

Sravan Kumar Amaragonda - G00473077

March 2025

1 Introduction

1.1 Application Selection

For this vulnerability assessment project, I selected Breakableflask, a deliberately vulnerable Flask web application designed for security training. Breakableflask was chosen because it contains multiple real-world vulnerabilities in a compact codebase, making it ideal for in-depth analysis. The application is a simple web service with multiple endpoints that demonstrate common security flaws. I focused on three critical vulnerabilities:

- OS Command Injection in the DNS lookup functionality
- Server-Side Template Injection (SSTI) in the greeting feature
- Python Pickle Deserialization in cookie handling

These vulnerabilities represent real attack vectors commonly exploited in production applications, making them highly relevant for practical security learning.

1.2 Project Objectives

The objectives of this assessment were to:

- Identify and understand the root causes of each vulnerability
- Demonstrate practical exploitation in a safe, controlled environment
- Implement secure coding fixes using industry best practices
- Validate fixes through testing and static analysis tools
- Document the complete security assessment process

2 Threat Analysis

2.1 OWASP Top 10 Mapping

The identified vulnerabilities map to the OWASP Top 10 Web Application Security Risks:

Vulnerability	**OWASP Category**	**Severity**	**Attack Vector**
OS Command Injection	A03: Injection	Critical	User input in system commands
SSTI	A03: Injection	High	User input in template rendering
Pickle Deserialization	A08: Software and Data Integrity Failures	Critical	Malicious serialized objects

Table 1: Security Vulnerability Overview

A03: Injection flaws occur when untrusted data is sent to an interpreter as part of a command or query. Both OS Command Injection and SSTI fall into this category as they allow attackers to inject malicious code that gets executed by the application.

A08: Software and Data Integrity Failures covers vulnerabilities in deserialization where untrusted data can lead to remote code execution. The Pickle vulnerability represents this risk.

2.2 Threat Modeling

Each vulnerability presents specific threats:

OS Command Injection

- **Threat:** Remote code execution on the server
- **Impact:** Complete system compromise, data theft, malware installation
- **Likelihood:** High (common vulnerability in poorly validated input)

Server-Side Template Injection (SSTI)

- **Threat:** Template code injection leading to information disclosure and remote code execution
- **Impact:** Access to application secrets, configuration data, potential server takeover
- **Likelihood:** Medium (requires template engine misuse)

Pickle Deserialization

- **Threat:** Arbitrary code execution through malicious serialized objects
- **Impact:** Complete application compromise, persistent backdoors
- **Likelihood:** High (when accepting serialized data from users)

3 Vulnerabilities Identified

3.1 OS Command Injection

The DNS lookup functionality takes user input and directly concatenates it into a system command without validation:

Listing 1: Vulnerable Flask Lookup Route

```
@app.route('/lookup', methods = ['POST', 'GET'])
def lookup():
    address = None
    if request.method == 'POST':
        address = request.form['address']
    return """
    <html>
       <body>""" + "Result:\n<br>\n" + (rp("nslookup " + address).replace('\n'
           , '\n<br>') if address else "") + """
          <form action = "/lookup" method = "POST">
             <p><h3>Enter address to lookup</h3></p>
             <p><input type = 'text' name = 'address'/></p>
             <p><input type = 'submit' value = 'Lookup'/></p>
          </form>
       </body>
    </html>
    """
```

- User input (`address`) is directly concatenated into the command string
- No input validation or sanitization
- Uses `popen()` which executes commands through a shell
- Shell metacharacters such as `&&`, `;`, `|` allow command chaining

3.2 Server-Side Template Injection (SSTI)

The greeting functionality inserts user input into a template string before rendering:

Listing 2: Server-Side Template Injection (SSTI)

```
@app.route('/sayhi', methods = ['POST', 'GET'])
def sayhi():
   name = ''
   if request.method == 'POST':
      name = '<br>Hello %s!<br><br>' %(request.form['name'])

   template = """
   <html>
      <body>
         <form action = "/sayhi" method = "POST">
            <p><h3>What is your name?</h3></p>
            <p><input type = 'text' name = 'name'/></p>
            <p><input type = 'submit' value = 'Submit'/></p>
         </form>
      %s
      </body>
   </html>
   """ %(name)
   return render_template_string(template)
```

- User input is embedded in the template string **before** rendering
- Jinja2 template syntax {{ }} in user input gets evaluated
- Allows access to Python objects and methods through the template context
- Can escalate to remote code execution

3.3 Pickle Deserialization

The application deserializes user-controlled cookie data using Python's pickle module:

Listing 3: Pickle Deserialization

```


@app.route('/cookie', methods = ['POST', 'GET'])
def cookie():
    cookieValue = None
    value = None

    if request.method == 'POST':
        cookieValue = request.form['value']
        value = cookieValue
    elif 'value' in request.cookies:
        cookieValue = pickle.loads(b64decode(request.cookies['value']))


    form = """
```

```
    <html>
       <body>Cookie value: """ + str(cookieValue) +"""
          <form action = "/cookie" method = "POST">
             <p><h3>Enter value to be stored in cookie</h3></p>
             <p><input type = 'text' name = 'value'/></p>
             <p><input type = 'submit' value = 'Set Cookie'/></p>
          </form>
       </body>
    </html>
    """
    resp = make_response(form)

    if value:
        resp.set_cookie('value', b64encode(pickle.dumps(value)))

    return resp
```

- Pickle can execute arbitrary code during deserialization
- Cookies are completely controlled by users
- The `__reduce__` method in objects can specify code to execute
- No validation of deserialized data

4 Proof-of-Concept Exploits

4.1 OS Command Injection Exploit

Method:

The vulnerability was tested by injecting additional commands using the `&&` operator.

Test 1: `google.com && dir`

Result: DNS lookup executed **and** directory listing displayed

Test 2: `google.com && whoami`

Result: Displayed current user account

Outcome: Successfully demonstrated remote command execution. In a real attack, this could be used to install malware, steal data, or create backdoors.

4.2 SSTI Exploit

Method:

Jinja2 template syntax was injected to test code execution.

Test 1: {{7*7}}

Result: Displayed 49 instead of literal text

Test 2: {{7*7+7}} **Result:** Displayed 56

Outcome: Successfully executed Python code through template injection. This could escalate to accessing {{config}} for secrets or achieving remote code execution through Python object introspection.

4.3 Pickle Deserialization Exploit

Method:

A Python script was created to generate a malicious pickled object:

Listing 4: Malicious Pickle Payload Generator

```
import pickle
import base64
import os
```

```
class Exploit(object):
    def __reduce__(self):
        return (os.system, ('echo You got hacked! > pwned.txt',))

malicious_pickle = pickle.dumps(Exploit())
payload = base64.b64encode(malicious_pickle).decode()
```

The script sent this payload as a cookie to the application. When the application unpickled it, the embedded command executed.

Outcome: Successfully created the `pwned.txt` file on the server, proving arbitrary code execution. This demonstrates complete application compromise.

5 Secure Coding Fixes

5.1 OS Command Injection Fix

Solution: Use the `subprocess` module with argument lists and input validation.

Fixed Code:

Listing 5: Secure SSTI Fix for OS Command Injection

```
import subprocess

@app.route('/lookup', methods=['POST','GET'])
def lookup():
    address = None
    result = ""

    if request.method == 'POST':
        address = request.form['address']

        try:
            # Validate input: only allow valid domain/IP format
            if address and all(c.isalnum() or c in '.-_' for c in address):
                # Use subprocess with argument list (NOT shell=True)
                result = subprocess.run(
                    ['nslookup', address],
                    capture_output=True,
                    text=True,
                    timeout=5
                ).stdout
            else:
                result = "Invalid input. Only alphanumeric characters, dots,
                    hyphens, and underscores allowed."
        except subprocess.TimeoutExpired:
            result = "Lookup timed out"
        except Exception as e:
            result = f"Error: {str(e)}"

    return result
```

Why This Works:

- Input validation restricts dangerous characters.
- Argument lists prevent shell command injection.
- `shell=True` is not used, so shell metacharacters are ignored.
- Timeout prevents resource exhaustion attacks.

5.2 SSTI Fix

Solution: Pass user input as template variables instead of embedding them directly into template strings.

Fixed Code:

Listing 6: Secure SSTI Fix for /sayhi

```
@app.route('/sayhi', methods = ['POST', 'GET'])
def sayhi():
    name = ''
    if request.method == 'POST':
        # SECURE FIX: Pass user input as variable, not in template string
        name = request.form['name']

        # Escape HTML to prevent XSS as well
        from markupsafe import escape
        name = escape(name)

    template = """
    <html>
        <body>
            <form action="/sayhi" method="POST">
                <p><h3>What is your name?</h3></p>
                <p><input type="text" name="name"/></p>
                <p><input type="submit" value="Submit"/></p>
            </form>
            {% if name %}
            <br>Hello {{ name }}!<br><br>
            {% endif %}
        </body>
    </html>
    """

    # Pass name as a variable to the template, not embedded in string
    return render_template_string(template, name=name)
```

Why This Works:

- User input is passed as a variable rather than embedded into template code.
- Jinja2 auto-escapes {{ `name` }} preventing template execution.
- Input is not interpreted as template code.
- HTML escaping adds an extra layer of defense against XSS.

5.3 Pickle Deserialization Fix

Solution: Replace unsafe `pickle` deserialization with JSON-based serialization.

Fixed Code:

Listing 7: Secure Cookie Handling Using JSON

```
import json

@app.route('/cookie', methods = ['POST', 'GET'])
def cookie():
    cookieValue = None
    value = None

    if request.method == 'POST':
        cookieValue = request.form['value']
        value = cookieValue
```

```
    elif 'value' in request.cookies:
        # SECURE FIX: Use JSON instead of pickle
        try:
            cookieValue = json.loads(request.cookies['value'])
        except:
            cookieValue = "Invalid cookie data"

    form = """
    <html>
       <body>Cookie value: """ + str(cookieValue) +"""
          <form action = "/cookie" method = "POST">
             <p><h3>Enter value to be stored in cookie</h3></p>
             <p><input type = 'text' name = 'value'/></p>
             <p><input type = 'submit' value = 'Set Cookie'/></p>
          </form>
       </body>
    </html>
    """
    resp = make_response(form)

    if value:
        # Store as JSON string instead of pickle
        resp.set_cookie('value', json.dumps(value))

    return resp
```

Why This Works:

- JSON only processes data, not executable code.
- Prevents arbitrary code execution during deserialization.
- Safe for untrusted user input.
- Industry-standard secure serialization format.

6 Security Testing and Exploit Automation

6.1 Bandit Static Analysis

Tool Used: Bandit v1.7.5 – Python security linter

Process:

- Ran initial scan on vulnerable code using `bandit -r main.py`
- Identified 10 security issues (3 high, 3 medium, 4 low)
- Implemented fixes based on Bandit findings
- Re-ran scan on fixed code
- Result: 7 issues remaining (2 high, 2 medium, 3 low) – approximately 30% reduction

Key Findings:

- Flagged unsafe `pickle` import and usage
- Identified `popen()` usage for command execution
- Warned about `subprocess` usage (false positive after fix)
- Confirmed elimination of assigned vulnerabilities

How It Helped:

- Automated discovery of security issues
- Confirmed vulnerability locations with line numbers
- Validated that fixes addressed root causes
- Provided confidence ratings for findings

6.2 Exploit Automation

Script Created: `pickle_exploit.py`

This script automated the pickle deserialization attack by generating malicious pickled objects, encoding payloads in Base64 format, sending crafted cookies to the application, and verifying successful exploitation.

Benefits:

- Repeatable testing process
- Proof of exploitability
- Useful for regression testing after fixes

7 Conclusion

This vulnerability assessment successfully identified, exploited, and remediated three critical security vulnerabilities in the BreakableFlask application. The project demonstrated the practical application of OWASP Top 10 framework, effective use of static analysis tools (Bandit), real-world exploitation techniques in a safe environment, implementation of secure coding practices, and verification through testing and rescanning.

The 30% reduction in Bandit findings and successful blocking of all three exploit attempts confirms the effectiveness of the applied fixes. This hands-on experience provided valuable insights into both offensive and defensive security practices, emphasizing the importance of secure coding from the design phase.

Additional Improvements

Beyond the three assigned vulnerabilities, the following additional issues were also addressed:

- `eval()` vulnerability in the expression evaluator (functionality disabled)
- SQL Injection in the product listing (added input validation and parameterized queries)

These additional fixes demonstrate a comprehensive security mindset and a defense-in-depth approach.

8 References

1. OWASP Top 10 Web Application Security Risks (2021). Available at: `https://owasp.org/Top10/`
2. Bandit Documentation. Available at: `https://bandit.readthedocs.io/`
3. Python Pickle Security. Available at: `https://docs.python.org/3/library/pickle.html`
4. BreakableFlask Repository. Available at: `https://github.com/stephenbradshaw/breakableflask`
5. Fixed Code Git Hub Link: https://github.com/Sravankumar3/breakableflask