Common Python Automation Vulnerabilities: Descriptions, Examples, and Mitigations

1. Path Traversal

Description:
Path traversal allows attackers to access files or directories outside the intended scope by manipulating file paths.

Vulnerable Example:
Suppose an attacker enters `../../etc/passwd` as input. This code would read sensitive system files:

```
import os

def read_file(filename):
    with open(f"/var/data/{filename}", "r") as f:
        return f.read()

# User input: ../../etc/passwd
print(read_file(input("Filename: ")))  # Attacker reads /etc/passwd!
```

Mitigated Example:

```
import os

def read_file(filename):
    base_dir = "/var/data/"
    full_path = os.path.abspath(os.path.join(base_dir, filename))
    if not full_path.startswith(os.path.abspath(base_dir)):
        raise ValueError("Invalid file path!")
    with open(full_path, "r") as f:
        return f.read()
```

2. Shell Injection (Command Injection)

Description:
Shell injection occurs when untrusted input is passed to system shell commands, allowing execution of arbitrary code.

Vulnerable Example:
User input like `; rm -rf /` could delete system files:

```
import os

def list_files(dir_name):
    os.system(f"ls {dir_name}")

# User input: ; rm -rf /
list_files(input("Directory: "))  # Attacker deletes all files!
```

Mitigated Example:

```
import subprocess

def list_files(dir_name):
    subprocess.run(["ls", dir_name], check=True)  # Arg list prevents
injection

list_files(input("Directory: "))
```

---

3. Insecure Deserialization

Description:
Untrusted data deserialized with formats like pickle may execute arbitrary code during loading.

Vulnerable Example:
Attacker crafts malicious pickle data that runs code on deserialization:

```
import pickle

data = input("Enter serialized object: ")
obj = pickle.loads(data)  # Arbitrary code execution possible!
```

Mitigated Example:

```
import json

data = input("Enter serialized object: ")
obj = json.loads(data)
```

---

4. Insecure Use of eval()/exec()

Description:
Using eval() or exec() on untrusted input can execute arbitrary Python code.

Vulnerable Example:

User input `__import__('os').system('rm -rf /')` is executed:

```
def calculate(expr):
    return eval(expr)  # Dangerous if expr is user-controlled

# User input: __import__('os').system('rm -rf /')
print(calculate(input("Expression: ")))  # Attacker deletes all files!
```

Mitigated Example:

```
import ast

def calculate(expr):
    node = ast.parse(expr, mode='eval')
    for subnode in ast.walk(node):
        if not isinstance(subnode, (ast.Expression, ast.BinOp, ast.Num,
ast.UnaryOp, ast.operator)):
            raise ValueError("Unsafe expression!")
    return eval(expr, {"__builtins__": {}})

print(calculate(input("Expression: ")))
```

---

5. Insecure Temporary File Handling

Description:

Improperly creating temp files can allow attackers to replace files or read their contents.

Vulnerable Example:

Attacker creates a symlink at `/tmp/mytempfile` before the program writes to it:

```
filename = "/tmp/mytempfile"
with open(filename, "w") as f:
    f.write("data")
# Attacker could create a symlink here to another file!
```

Mitigated Example:

```
import tempfile

with tempfile.NamedTemporaryFile(delete=True) as f:
    f.write(b"data")
    f.flush()
```

6. Hardcoded Credentials/Secrets

Description:
Storing secrets in code can lead to accidental leaks if the code is shared publicly.

Vulnerable Example:
If code is pushed to a public repo, the API key is exposed:

```
API_KEY = "123456789abcdef"  # Hardcoded secret

def get_data():
    pass  # Uses API_KEY
```

Mitigated Example:

```
import os

API_KEY = os.environ.get("API_KEY")
def get_data():
    pass  # Uses API_KEY
```

7. XML External Entity (XXE) Attacks

Description:
Processing untrusted XML with unsafe parsers allows attackers to read files or perform network requests.

Vulnerable Example:
A malicious XML input accesses internal files:

```
import xml.etree.ElementTree as ET

data = input("XML data: ")
root = ET.fromstring(data)  # XXE possible!
# Attacker can read /etc/passwd via crafted XML
```

Mitigated Example:

```
import defusedxml.ElementTree as ET

data = input("XML data: ")
root = ET.fromstring(data)  # defusedxml disables XXE
```

8. SQL Injection

Description:
Concatenating user input into SQL queries allows attackers to alter the query logic and access unauthorized data.

Vulnerable Example:
User input `' OR 1=1 --` returns all users:

```
import sqlite3

conn = sqlite3.connect("example.db")
user = input("Username: ")
cursor = conn.execute(f"SELECT * FROM users WHERE username = '{user}'")
# User input: ' OR 1=1 --
```

Mitigated Example:

```
import sqlite3

conn = sqlite3.connect("example.db")
user = input("Username: ")
cursor = conn.execute("SELECT * FROM users WHERE username = ?", (user,))
```

---

9. Cross-Site Scripting (XSS)

Description:
If user input is shown in an HTML context without escaping, attackers can inject scripts.

Vulnerable Example:
User input `<script>alert('XSS')</script>` executes JavaScript in the browser:

```
def render(user_input):
    return f"<div>{user_input}</div>"

# Output sent to web: <div><script>alert('XSS')</script></div>
print(render(input("Enter text: ")))
```

Mitigated Example:

```
import html

def render(user_input):
    return f"<div>{html.escape(user_input)}</div>"
```

---

10. Insecure Permissions (Privilege Escalation)

Description:
Running scripts with more privileges than needed can turn minor bugs into critical exploits.

Vulnerable Example:
Running as root, any bug in the script can damage the system:

```
# Script run as root
import os

os.system("touch /etc/important_config")
```

Mitigated Example:

```
# Run script as a regular user, use least privileges necessary.
```

---

11. Race Conditions

Description:
Race conditions happen when a program's behavior changes due to the timing of external events, often leading to security issues.

Vulnerable Example:
Attacker replaces the file between the existence check and open:

```
import os

filename = "/tmp/data.txt"
if not os.path.exists(filename):
    with open(filename, "w") as f:
        f.write("data")
# TOCTOU allows attacker to swap the file
```

Mitigated Example:

```
import os

flags = os.O_CREAT | os.O_EXCL | os.O_WRONLY
try:
    fd = os.open("/tmp/data.txt", flags)
    with os.fdopen(fd, "w") as f:
```

```
        f.write("data")
except FileExistsError:
    print("File already exists.")
```

---

12. Unvalidated Input

Description:
Accepting and using user input without validation can lead to logic errors or vulnerabilities.

Vulnerable Example:
A negative age or string is accepted:

```
def process(age):
    print(f"You are {age} years old.")

# User input: -999 or "hello"
process(input("Age: "))  # Invalid/unexpected input accepted
```

Mitigated Example:

```
def process(age):
    age = int(age)
    if not (0 < age < 150):
        raise ValueError("Invalid age")
    print(f"You are {age} years old.")

process(input("Age: "))
```

---

13. Resource Injection

Description:
Allowing untrusted input to dictate resource names (like filenames) can result in access to unauthorized resources.

Vulnerable Example:
User provides `/etc/passwd` as filename:

```
def print_file(filename):
    with open(filename, "r") as f:
        print(f.read())

# User input: /etc/passwd
print_file(input("File to print: "))  # Attacker reads system files!
```

Mitigated Example:

```
import os

def print_file(filename):
    allowed_files = {"file1.txt", "file2.txt"}
    if filename not in allowed_files:
        raise ValueError("File not allowed")
    with open(filename, "r") as f:
        print(f.read())
```

---

14. Information Disclosure

Description:
Logging or exposing sensitive data (like passwords or tokens) can leak secrets to attackers.

Vulnerable Example:
User's password is shown in logs:

```
def login(user, password):
    try:
        # ... authentication logic
        raise Exception("Failed login")
    except Exception as e:
        print(f"Error: {e}, user: {user}, password: {password}")  #
Sensitive info leaked!
```

Mitigated Example:

```
def login(user, password):
    try:
        # ... authentication logic
        raise Exception("Failed login")
    except Exception as e:
        print("Login failed.")  # No sensitive info revealed
```

---

15. Use of Outdated Libraries

Description:
Using dependencies with known vulnerabilities exposes your code to risks present in those libraries.

Vulnerable Example:
Old `requests` library version with CVEs is used:

```
# No version pinning or updates
import requests
# Vulnerabilities in old version may be exploited
```

Mitigated Example:

```
# Use requirements.txt and keep dependencies up to date
# Example: requests>=2.31.0
```

---

16. Server-Side Request Forgery (SSRF)

Description:
SSRF allows attackers to make your server send requests to internal or protected resources.

Vulnerable Example:
User inputs `http://localhost:8000/admin` to access internal endpoints:

```
import requests

url = input("Enter URL: ")
resp = requests.get(url)
# Attacker accesses internal-only endpoints!
```

Mitigated Example:

```
import requests
from urllib.parse import urlparse

def safe_request(url):
    parsed = urlparse(url)
    if parsed.hostname in ["localhost", "127.0.0.1"]:
        raise ValueError("Refusing to access internal addresses")
    resp = requests.get(url)
    return resp.text

print(safe_request(input("Enter URL: ")))
```

---

17. Unsafe YAML Loading

Description:
Unsafe YAML loading (`yaml.load`) can execute arbitrary code embedded in YAML input.

Vulnerable Example:
Attacker crafts YAML that runs code during loading:

```
import yaml

data = input("YAML: ")
obj = yaml.load(data)  # Unsafe: arbitrary code execution
```

Mitigated Example:

```
import yaml

data = input("YAML: ")
obj = yaml.safe_load(data)
```

---

18. Unrestricted File Upload

Description:
Allowing users to upload any file type or with any name can result in malicious files being stored or executed.

Vulnerable Example:
User uploads a `.php` shell or replaces system files:

```
def upload(file):
    with open(f"/uploads/{file.filename}", "wb") as f:
        f.write(file.read())
# User uploads .php shell script as file
```

Mitigated Example:

```
import os

def upload(file):
    allowed_ext = {".jpg", ".png", ".txt"}
    name, ext = os.path.splitext(file.filename)
    if ext not in allowed_ext:
        raise ValueError("File type not allowed")
    safe_name = os.path.basename(file.filename)
    with open(f"/uploads/{safe_name}", "wb") as f:
        f.write(file.read())
```

---

19. Unsafe Pickle Usage

Description:
Using `pickle.loads` on untrusted input can execute attacker-supplied code.

Vulnerable Example:
Attacker submits pickle data triggering code execution:

```
import pickle

data = input("Pickle: ")
obj = pickle.loads(data)  # Arbitrary code execution!
```

Mitigated Example:

```
import json

data = input("Data: ")
obj = json.loads(data)
```

Common Vulnerabilities When Making API Calls from Python Scripts

When integrating APIs in Python scripts, be aware of several security vulnerabilities that can arise, especially when handling external input or sensitive data.

---

1. Injection Attacks (Including SSRF)

Description:
If user input is used to construct API endpoints or headers, attackers can manipulate requests, cause Server-Side Request Forgery (SSRF), or inject malicious data.

Vulnerable Example:
Suppose a user enters `http://localhost/admin` or `http://169.254.169.254/latest/meta-data/`. The script will make a request to an internal resource.

```
import requests
url = input("Enter API URL: ")
response = requests.get(url)  # User may supply internal or sensitive URL
```

Mitigated Example:

```
from urllib.parse import urlparse
import requests

def safe_get(url):
    parsed = urlparse(url)
    if parsed.hostname in ["localhost", "127.0.0.1", "169.254.169.254"]:
        raise ValueError("Blocked internal address")
    return requests.get(url)

safe_get(input("Enter API URL: "))
```

---

2. Header Injection

Description:
Unsanitized input in HTTP headers can allow attackers to inject additional headers or manipulate requests.

Vulnerable Example:
If a user enters a value like `evil\r\nX-Injected-Header: injected`, it could corrupt the HTTP request.

```
import requests
user_agent = input("User-Agent: ")
requests.get("https://api.example.com", headers={"User-Agent": user_agent})
```

Mitigated Example:

```
import requests
import re
user_agent = input("User-Agent: ")
if not re.match(r"^[a-zA-Z0-9 ._-]+$", user_agent):
    raise ValueError("Invalid User-Agent")
requests.get("https://api.example.com", headers={"User-Agent": user_agent})
```

---

3. Sensitive Data Exposure

Description:
Logging or storing API keys, tokens, or sensitive payloads can leak secrets if logs or files are accessed by unauthorized users.

Vulnerable Example:
API key is printed in logs, which could be accessed by attackers.

```

```

```
api_key = "supersecret"
response = requests.get("https://api.example.com", headers=
{"Authorization": f"Bearer {api_key}"})
print(response.request.headers)  # Logging sensitive headers
```

Mitigated Example:

```
import os
api_key = os.environ.get("API_KEY")
response = requests.get("https://api.example.com", headers=
{"Authorization": f"Bearer {api_key}"})
# Do not log or expose headers containing secrets
```

---

4. Improper Certificate Validation

Description:
Disabling SSL verification (`verify=False`) exposes the script to Man-in-the-Middle (MitM) attacks.

Vulnerable Example:
Anyone on the network could intercept/modify traffic.

```
import requests
requests.get("https://api.example.com", verify=False)  # SSL not verified!
```

Mitigated Example:

```
import requests
requests.get("https://api.example.com")  # SSL verification enabled by
default
```

---

5. Unvalidated/Unsanitized Input

Description:
Passing untrusted input directly in query parameters or request bodies can result in malformed requests or injection attacks.

Vulnerable Example:
If user input contains special characters, it may corrupt the request or cause unintentional effects.

```
import requests
param = input("Parameter: ")
```

```
requests.get(f"https://api.example.com/data?param={param}")
```

Mitigated Example:

```
import requests
param = input("Parameter: ")
response = requests.get("https://api.example.com/data", params={"param":
param})
```

---

6. Improper Error Handling and Information Disclosure

Description:
Verbose error messages or stack traces can leak sensitive data or internal implementation details.

Vulnerable Example:
An API error prints the full exception, potentially leaking sensitive details.

```
try:
    response = requests.get("https://api.example.com/data")
    response.raise_for_status()
except Exception as e:
    print(e)  # May expose sensitive info
```

Mitigated Example:

```
try:
    response = requests.get("https://api.example.com/data")
    response.raise_for_status()
except Exception:
    print("An error occurred while fetching data.")
    # Log details securely for internal troubleshooting only
```

---

7. Insecure Deserialization

Description:
Unsafe deserialization (e.g., using `pickle.loads` on API data) can lead to arbitrary code execution.

Vulnerable Example:
If the API response is attacker-controlled, code execution may occur.

```
import pickle
data = requests.get("https://api.example.com/data").content
```

```
obj = pickle.loads(data)  # Dangerous if data is attacker-controlled
```

Mitigated Example:

```
import json
data = requests.get("https://api.example.com/data").text
obj = json.loads(data)
```

---

8. Race Conditions and Replay Attacks

Description:
Concurrent or repeated API calls without proper checks can result in inconsistent state or replay of actions.

Vulnerable Example:
If a script is run multiple times, it could create duplicate transactions.

```
import requests
token = "TXN123"
for _ in range(2):
    requests.post("https://api.example.com/payment", json={"token": token})
# May process the same payment twice (replay attack)
```

Mitigated Example:

- Use idempotency keys in API requests.
- Ensure that backends handle duplicate requests safely.

---

9. Insecure Dependency Usage

Description:
Using outdated libraries (like `requests`) may expose known vulnerabilities.

Vulnerable Example:

```
# Using requests==2.3.0, which may have unpatched CVEs
import requests
```

Mitigated Example:

- Use the latest versions and update dependencies frequently.
- Use tools like `pip list --outdated` or `pip-audit` to monitor dependency health.

---

10. Improper Authorization/Authentication Handling

Description:
Failing to validate tokens or using over-privileged tokens can grant attackers unauthorized access.

Vulnerable Example:
Using an admin token for all actions, even those that don't require it.

```
admin_token = "adminsecrettoken"
requests.get("https://api.example.com/data", headers={"Authorization":
f"Bearer {admin_token}"})
```

Mitigated Example:

- Use the principle of least privilege: only use tokens with minimal required permissions.
- Validate tokens on every request and check expiration.

---

11. Denial of Service (DoS)

Description:
Accepting or processing very large API responses or payloads can exhaust script or server resources.

Vulnerable Example:
Requests an unbounded file and reads it all into memory.

```
import requests
response = requests.get("https://api.example.com/large-data")
data = response.content  # May consume excessive memory
```

Mitigated Example:

```
import requests
response = requests.get("https://api.example.com/large-data", stream=True,
timeout=10)
for chunk in response.iter_content(chunk_size=8192):
    # Process chunk
    pass
```

Additional Best Practices

- **Timeouts:** Always set reasonable timeouts for API requests.
- **Retry Logic:** Handle transient failures with proper retry strategies.
- **Input Validation:** Always validate and sanitize user input before using it in requests.

Summary:
When making API calls from Python, always validate and sanitize input, handle secrets securely, update dependencies, and avoid exposing sensitive information via logs or error messages.