Prerequisites for the Backdoor (Reverse Shell) Project

By: **Inlighn Tech**

Overview

The Backdoor (Reverse Shell) project requires students to develop a pair of Python scripts: client.py (the backdoor that connects to a server and executes commands) and server.py (the listener that controls the client). Together, they establish a reverse shell, where the client initiates a connection to the server, allowing remote command execution and file transfer. The solution uses socket for networking, json for data serialization, subprocess for command execution, os for directory changes, and base64 for file encoding, with retry logic for connection stability. To write these scripts independently, students must master several key concepts and skills from the curriculum. These prerequisites, paired with practical Python code examples, ensure students understand how reverse shells work and can implement both components effectively.

Prerequisite Knowledge and Skills

1. **Networking Basics – Sockets and Reverse Shells**
    - **What to Know**:
        - Sockets enable network communication; AF_INET uses IPv4, SOCK_STREAM uses TCP.
        - A reverse shell has the target (client) connect to the attacker (server), unlike a bind shell where the server listens first.
        - IP addresses and ports define connection endpoints (e.g., 192.168.1.102:4444).
    - **How It Applies**:
        - server.py sets up a listener with socket.bind() and listen(), while client.py connects with socket.connect().
    - **Curriculum Source**:
        - "NETWORKING, SOCKETS, AND CYBERSECURITY" PDFs (e.g., "Introduction to Sockets.pdf").
    - **Practical Prep**:
        - Test socket communication manually with nc -l 4444 (server) and nc 127.0.0.1 4444 (client).
    - **Python Code Block**:
    -

```
# Simple socket client-server test
import socket
import threading

def server():
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.bind(("127.0.0.1", 4444))
    s.listen(1)
    conn, addr = s.accept()
    print(f"Connected from {addr}")
    conn.close()

def client():
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.connect(("127.0.0.1", 4444))
    print("Connected to server")
    s.close()
```

```
t = threading.Thread(target=server)
t.start()
client()
t.join()
```

- **Run It**: Save as socket_test.py, run with python socket_test.py. Demonstrates basic client-server connection, preparing for server() and run().

2. **Socket Programming**
   - **What to Know**:
     - socket.bind(), listen(), and accept() set up a server; connect() initiates a client connection.
     - send() and recv() handle data transfer; SO_REUSEADDR allows port reuse.
     - Use loops for persistent connections (e.g., retry on failure).
   - **How It Applies**:
     - server.py listens and accepts connections; client.py retries connections and sends/receives data.
   - **Curriculum Source**:
     - "NETWORKING, SOCKETS, AND CYBERSECURITY" PDF ("Introduction to Sockets.pdf").
   - **Practical Prep**:
     - Test sending data over sockets.
   - **Python Code Block**:

```
# Send and receive data over sockets
import socket
import threading

def server():
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.bind(("127.0.0.1", 4444))
    s.listen(1)
    conn, _ = s.accept()
    conn.send("Hello from server".encode())
```

```
    conn.close()

def client():
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.connect(("127.0.0.1", 4444))
    data = s.recv(1024).decode()
    print(f"Received: {data}")
    s.close()

t = threading.Thread(target=server)
t.start()
client()
t.join()
```

-
    - **Run It**: Save as socket_data_test.py, run with python socket_data_test.py. Prepares for send() and recieve() functions.

3. **JSON Serialization**
    - **What to Know**:
        - json.dumps() converts Python objects to strings; json.loads() parses them back.
        - Ensures reliable data transfer over sockets (e.g., strings, lists).
    - **How It Applies**:
        - Both scripts use json to encode/decode commands and results between client and server.
    - **Curriculum Source**:
        - "WORKING WITH DATA STRUCTURES & FILE HANDLING" Lesson 8 (Working with JSON).
    - **Practical Prep**:
        - Test JSON encoding/decoding.
    - **Python Code Block**:
    -

```
# JSON encode and decode
import json

data = {"command": "dir"}
json_data = json.dumps(data)
print(f"Encoded: {json_data}")
decoded = json.loads(json_data)
```

```
print(f"Decoded: {decoded}")
```

- **Run It**: Save as json_test.py, run with python json_test.py. Prepares for send() and recieve().

4. **Command Execution with subprocess**
   - **What to Know**:
     - subprocess.Popen() runs shell commands, capturing stdout, stderr, and stdin.
     - shell=True enables command interpretation; universal_newlines=True returns text output.
   - **How It Applies**:
     - client.py uses subprocess to execute commands received from the server and send back results.
   - **Curriculum Source**:
     - Not explicitly listed; assumes basic Python knowledge (common in system automation).
   - **Practical Prep**:
     - Test command execution.
   - **Python Code Block**:

```
# Run a shell command
import subprocess

cmd = "dir" if os.name == "nt" else "ls -l"
process = subprocess.Popen(cmd, shell=True, stdout=subprocess.PIPE, stderr=subprocess.PIPE, universal_newlines=True)
output = process.stdout.read() + process.stderr.read()
print(f"Output: {output}")
```

- **Run It**: Save as subprocess_test.py, run with python subprocess_test.py. Prepares for command handling in run() (client).

5. **File Operations with os and File I/O**
    - **What to Know**:
        - os.chdir() changes directories; open() in binary mode ('rb', 'wb') reads/writes files.
        - Used for cd, download, and upload commands.
    - **How It Applies**:
        - client.py changes directories and handles file transfers; server.py manages file downloads/uploads.
    - **Curriculum Source**:
        - "PYTHON BASICS" Lesson 11 (IO with Basic Files).
        - "WORKING WITH DATA STRUCTURES & FILE HANDLING" Lesson 10 (Opening and Reading Files and Folders Python OS Module).
    - **Practical Prep**:
        - Test file and directory operations.
    - **Python Code Block**:
    -
```
# Change directory and read/write a file
import os

# Change directory
os.chdir(os.path.expanduser("~"))  # Home directory
print(f"Current directory: {os.getcwd()}")

# Write and read a file
with open("test.txt", "wb") as f:
    f.write(b"Test data")
with open("test.txt", "rb") as f:
    print(f"File content: {f.read()}")
```
    -
        - **Run It**: Save as file_test.py, run with python file_test.py. Prepares for cd, download, and upload.
6. **Base64 Encoding for File Transfer**
    - **What to Know**:
        - base64.b64encode() converts binary data to text; b64decode() reverses it.
        - Ensures safe transmission of files over text-based sockets.

- **How It Applies**:
  - Both scripts use base64 to encode/decode files for download and upload commands.
- **Curriculum Source**:
  - Not explicitly taught; assumes basic Python knowledge (common in networking projects).
- **Practical Prep**:
  - Test base64 encoding/decoding.
- **Python Code Block**:
-

```
# Encode and decode with base64
import base64

data = b"Test file content"
encoded = base64.b64encode(data).decode("utf-8")
print(f"Encoded: {encoded}")
decoded = base64.b64decode(encoded)
print(f"Decoded: {decoded}")
```

-

  - **Run It**: Save as base64_test.py, run with python base64_test.py. Prepares for file transfer encoding.

7. **Exception Handling and Retry Logic**
   - **What to Know**:
     - Use try/except to handle connection errors (e.g., ConnectionRefusedError) and JSON parsing errors (ValueError).
     - Implement retries with time.sleep() for robustness.
   - **How It Applies**:
     - client.py retries connections on failure; both scripts handle incomplete JSON data gracefully.
   - **Curriculum Source**:
     - "PYTHON BASICS" Lesson 21 (Errors and Exception Handling).
   - **Practical Prep**:
     - Test retry logic.
   - **Python Code Block**:
   -

```
# Retry on connection failure
import socket
import time

def connect_with_retry(ip, port):
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    while True:
        try:
            s.connect((ip, port))
            print("Connected")
            break
        except ConnectionRefusedError:
            print("Connection failed, retrying...")
            time.sleep(1)
    s.close()

connect_with_retry("127.0.0.1", 9999)  # No server running
```

- **Run It**: Save as retry_test.py, run with python retry_test.py. Prepares for server() in client.py.

How the Backdoor (Reverse Shell) Works

- **Concept**:
  - A reverse shell backdoor connects from the target (client) to the attacker (server), allowing remote command execution and file transfer.
  - This simulates real-world malware for educational purposes.
- **Client Workflow (client.py)**:
  - Connect to the server with retries on failure.
  - Receive commands, execute them (subprocess), handle cd, and transfer files (base64).
  - Send results back to the server.
- **Server Workflow (server.py)**:
  - Listen for a client connection and accept it.
  - Send commands via user input, handle file transfers, and display results.

    - Exit on exit command.
- **Why Reverse Shell?**:
    - Bypasses firewalls by having the client initiate the connection, common in penetration testing scenarios.

How to Write the Backdoor (Reverse Shell) Scripts

Using these prerequisites, students can build both scripts:

- **Client (client.py)**:
    1. Import socket, json, subprocess, os, base64, and time.
    2. Define server() to connect with retries.
    3. Write send() and recieve() with JSON encoding/decoding.
    4. Implement run() to handle commands (exit, cd, download, upload, others).
    5. Call server() and run() with hardcoded IP/port.
- **Server (server.py)**:
    1. Import socket, json, and base64.
    2. Define server() to bind, listen, and accept a connection.
    3. Write send() and recieve() with JSON handling.
    4. Implement run() to prompt for commands and process responses.
    5. Call server() and run() with hardcoded IP/port.

Notes for Students

- **Setup**: No external libraries needed beyond Python's standard library. Run server.py first on one machine (python server.py), then client.py on another or the same machine (python client.py), updating the IP (192.168.1.102) to your server's local IP. Use ifconfig (Linux) or ipconfig (Windows) to find it.
- **Engagement**: Run each code block to practice sockets, JSON, and file transfers. Test the full scripts with commands like whoami, cd dir, download file.txt, and upload file.txt.

- **Tips**: Test on a local network or loopback (127.0.0.1) to avoid legal/ethical issues; ensure no firewall blocks port 4444. Start server.py before client.py.

Jump to the Backdoor (Reverse Shell) Project Description if you are ready now.