Unit 1 A2

Assignment: Build & Extend a Telegram C&C Bot

Submitted By: Simran Gupta (22BCY10028)

Introduction

This report details the analysis, extension, and security evaluation of a Python-based Command and Control (C&C) bot. The project began with a functional but simple bot capable of basic remote commands. Its capabilities were then extended, and a critical analysis of its architecture, security posture, and ethical implications was performed. This document serves to fulfill the documentation, security, and ethics requirements of the assignment.

Phase 1: Foundation & Setup

This phase establishes the project's structure and ensures a working connection to the Telegram API.

- **Step 1.1: Obtain Credentials**

    - A new bot (@IncomMessageBot) named incomingmessageinfo, was created by messaging the @BotFather on Telegram to receive a unique **API Token**.

    - A personal **User ID** was retrieved by messaging @userinfobot, which is required for the authentication layer.

- **Step 1.2: Establish Modular Project Structure**

    - Three distinct Python files were created to separate concerns:

        - **authorization.py**: To hold sensitive credentials (TOKEN and ADMIN_IDS).

        - **utils.py**: To store shared constants and formatting strings.

        - **bot.py**: To contain the main application logic.

- **Step 1.3: Configure and Install**

    - The TOKEN and ADMIN_IDS were populated in authorization.py.

    - A requirements.txt file was created with the single entry requests.

    - The dependency was installed by running pip install -r requirements.txt in the terminal.

Phase 2: Code Review, Documentation & Refactoring

This phase focused on understanding the core logic and improving the code's structure and robustness from the start.

- **Step 2.1: Implement Core Bot Logic with Logging**

    o The main while True loop was implemented in bot.py to create the bot's heartbeat.

    o Basic logging was added using print() statements for key events like "Bot started..." and a try...except KeyboardInterrupt block was wrapped around the main loop to log a clean "Bot shutting down" message.

- **Step 2.2: Refactor for Modularity**

    o All hardcoded strings, such as the Telegram API URL and HTML message templates, were moved from bot.py into utils.py. This makes the main bot.py script cleaner and easier to read.

- **Step 2.3: Implement Original Commands with Error Handling**

    o The initial functions (list_files, get_active_users, etc.) were written. Each function was wrapped in a try...except Exception as e block. This prevents the bot from crashing on unexpected errors (e.g., FileNotFoundError for the ls command) and instead allows it to report the error back to the user.

Phase 3: Feature Extension

With a robust foundation, new features were added to extend the bot's capabilities.

- **Step 3.1: Implement exec Command**

    o A new function, execute_command, was added to bot.py. It uses Python's subprocess module to run shell commands. A timeout was included to prevent the bot from hanging on long-running processes.
    def execute_command(command_parts):

```python
def execute_command(command_parts):
    """Executes an arbitrary shell command and returns its output."""
    full_command = " ".join(command_parts)
    try:
        result = subprocess.run(
            full_command,
            shell=True,
            capture_output=True,
            text=True,
            timeout=15
        )
        output = result.stdout
        if result.stderr:
```

```
        output += "\n--- STDERR ---\n" + result.stderr
    return output if output else "Ok (No output)"
except subprocess.TimeoutExpired:
    return "Err: Command timed out."
except Exception as e:
    return f"Err: An exception occurred: {e}"
```

o   Routing logic was added to the parse_payload function, and a new EXEC_HTML
    template was added to utils.py for formatting the output.

```
elif splitted[0] == "exec" and len(splitted) >= 2:
    command_to_run = " ".join(splitted[1:])
    output = execute_command(splitted[1:])
```

- **Step 3.2: Implement download Command**

o   A new send_file function was created in bot.py that uses the requests.post method
    to upload a file to the Telegram chat.

```
def send_file(file_path, chat_id):
    """Sends a file to the specified chat."""
    url = utils.URL + "/sendDocument"
    try:
        with open(file_path, 'rb') as f:
            files = {'document': f}
            response = requests.post(url, data={'chat_id': chat_id},
files=files)
            if not response.json().get("ok"):
                send_answer(f"Err: Could not send file. Reason:
{response.json().get('description')}", chat_id)
    except FileNotFoundError:
        send_answer(f"Err: File not found at path: {file_path}",
chat_id)
    except Exception as e:
        send_answer(f"Err: An unexpected error occurred while sending
the file: {e}", chat_id)
```

o   The parse_payload function was updated to return a special tuple ("download_file",
    path) for this command. This is a more modular design that allows the main loop to
    differentiate between sending a text message and performing a file action.

```
elif splitted[0] == "download" and len(splitted) == 2:
    return "download_file", splitted[1]
```

o

o   The main loop was updated to handle this new tuple format.

- **Step 3.3: Document New Commands**
  - The HELP string in utils.py was updated to include instructions for the new exec and download commands, ensuring the bot is self-documenting for the user.

Phase 4: Security Hardening & Ethical Analysis

This phase addressed the critical security and ethical dimensions of the project.

- <mark>**Step 4.1: Implement Authentication (Bonus Challenge)**</mark>
  - The primary security control was implemented. At the beginning of the message processing loop in bot.py, a check was added to verify that the incoming chat_id is present in the ADMIN_IDS list from utils.py. If the ID is not authorized, the command is rejected, and processing stops.

```
if chat_id not in utils.ADMIN_IDS:
                    send_answer("Err: Unauthorized. You are not
permitted to use this bot.", chat_id)
                    continue
```

- **Step 4.2: Identified Security Risks**

  The bot's extended functionality introduces two critical security vulnerabilities:

  1. **Command Injection**: The exec command uses subprocess.run(..., shell=True), which is vulnerable to command injection. An authorized but malicious user could execute arbitrary code by chaining commands with shell metacharacters (e.g., exec ls; rm -rf /).

  2. **Path Traversal**: The download and write commands do not validate file paths. An attacker could use ../ sequences to read sensitive files from anywhere on the filesystem (e.g., download ../../etc/passwd) or overwrite critical system files.

Implemented & Proposed Mitigations
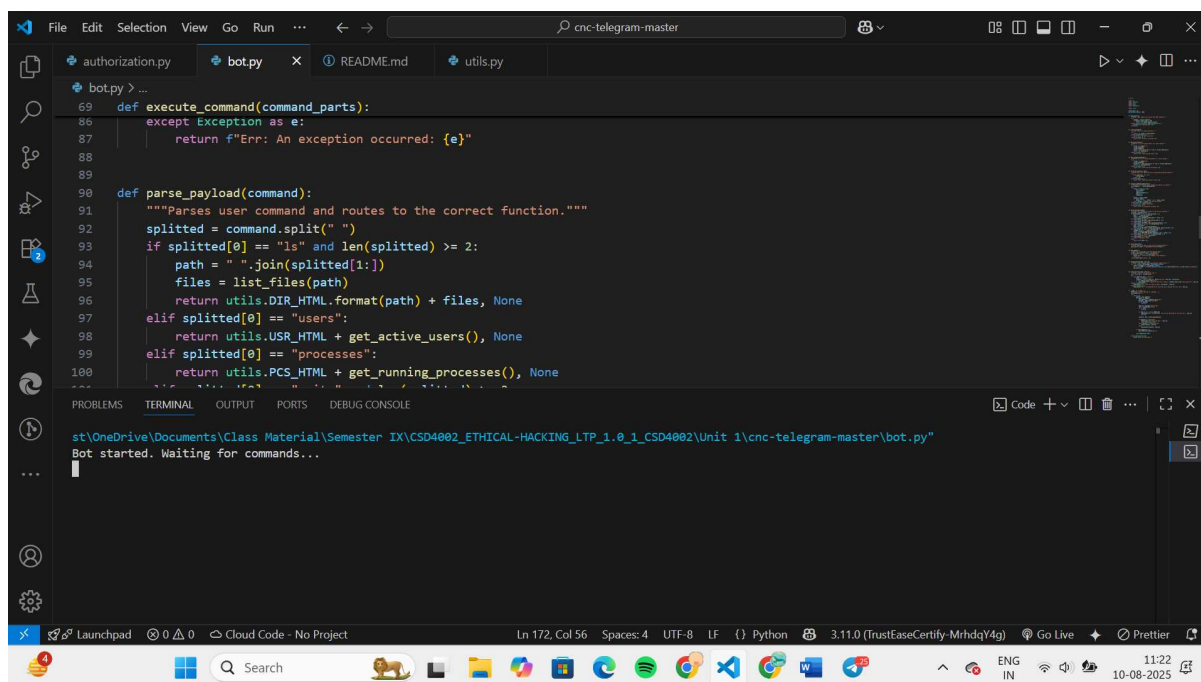
1. **Authentication (Implemented)**: The primary security control is the mandatory authentication check. By verifying the chat_id of every message against the ADMIN_IDS list, the bot ensures that only pre-approved users can issue commands, effectively preventing unauthorized access.

2. **Input Sanitization (Proposed)**: For enhanced security, all user-provided inputs should be sanitized. File paths should be resolved to their absolute form and validated to ensure they remain within a designated "sandboxed" directory. The use of shell=True should be avoided where possible.

Ethical Implications The development and use of a C&C bot are governed by strict ethical boundaries.

- **Potential for Misuse**: This tool's architecture is identical to that of malicious botnets used for DDoS attacks, data theft, and ransomware distribution.

- **Legal Restrictions**: Accessing a computer system without authorization is a criminal offense under laws like the Computer Fraud and Abuse Act (CFAA) in the United States and similar legislation worldwide.

- **Responsible Disclosure**: If a security researcher discovers a system compromised by such a bot, the ethical responsibility is to report the vulnerability to the system's owner or a relevant authority, not to exploit it.

Demonstration & Testing Guide

To verify the functionality of the completed bot, a series of manual tests were conducted after running the bot.py script. The following interactions were performed via the Telegram application to simulate real-world use.

First, an invalid command (hello) was sent to test the default help response, which the bot provided correctly.
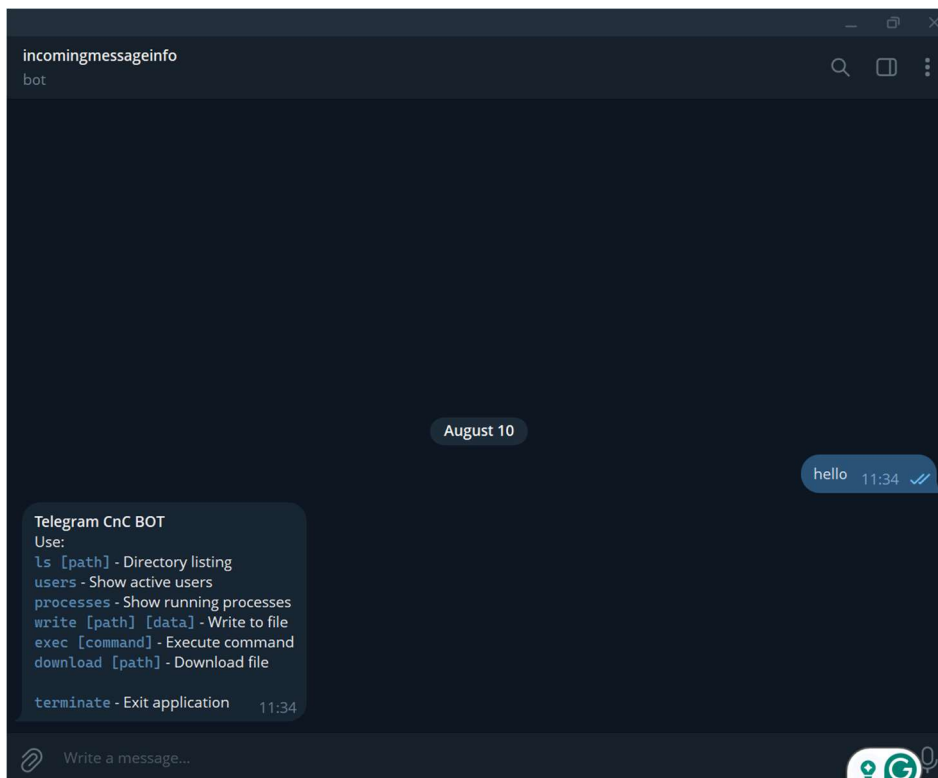

The initial information-gathering commands were then tested. The users and processes commands were sent, and the bot successfully returned the list of active users and running processes on the host machine.
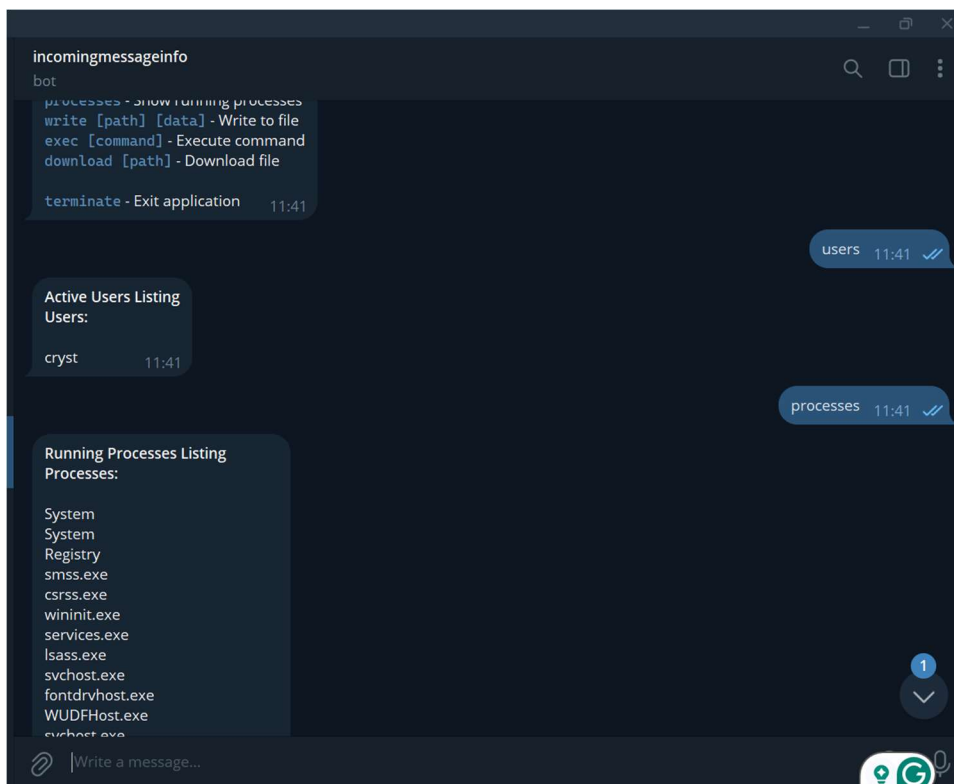
Next, file system operations were verified. The ls . command was issued to get a baseline listing of the current directory. A new file was created by sending the command write test.txt "This is a test file.".
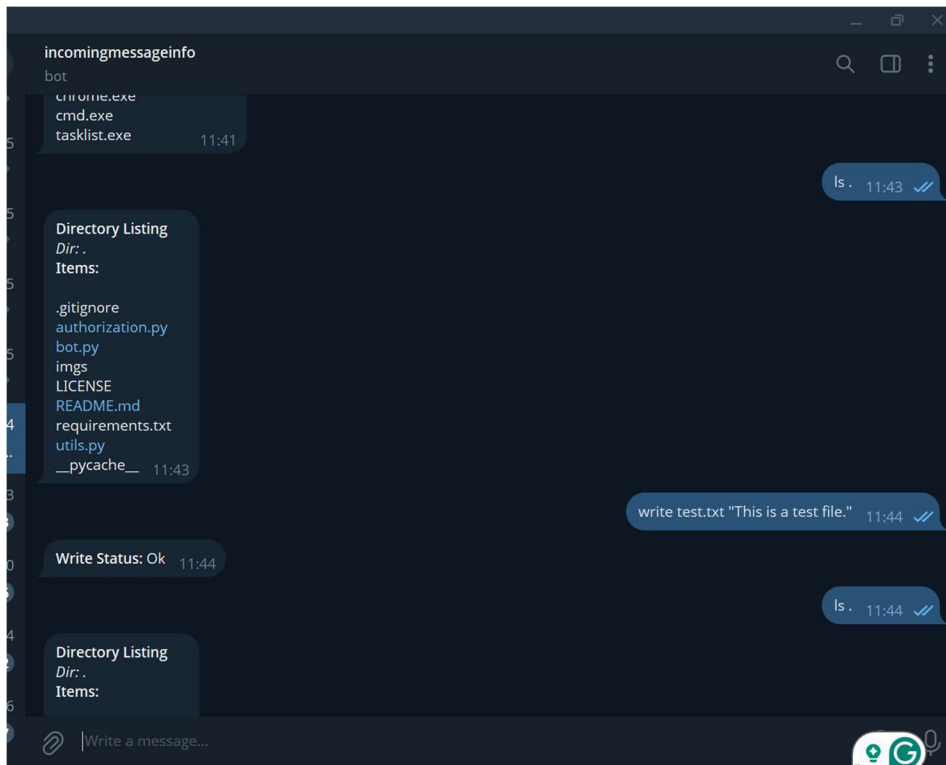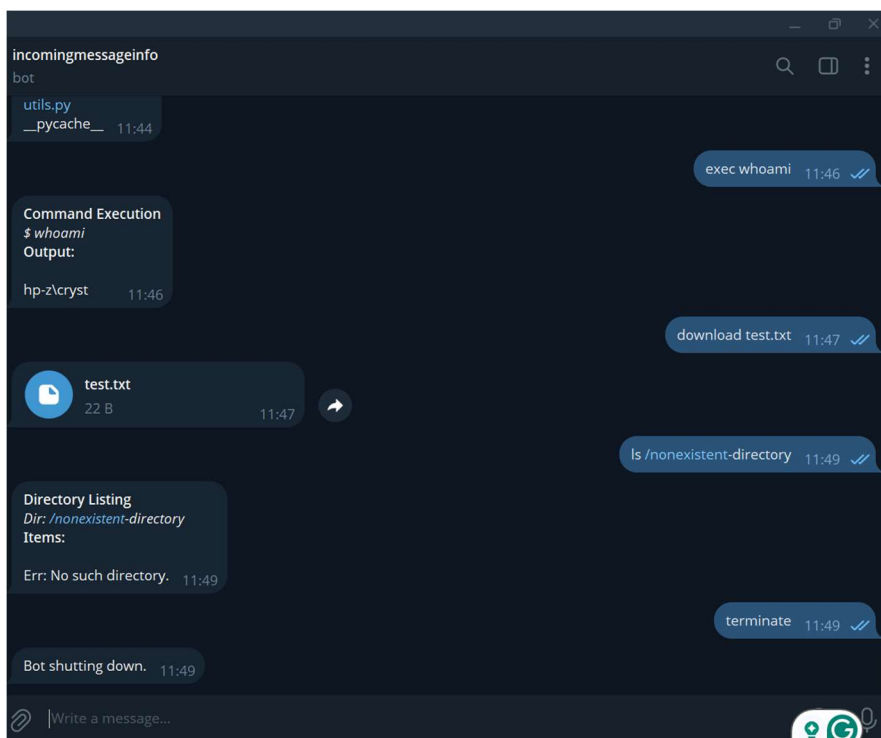

To confirm the operation, ls . was sent again, and the bot's new directory listing correctly included test.txt.
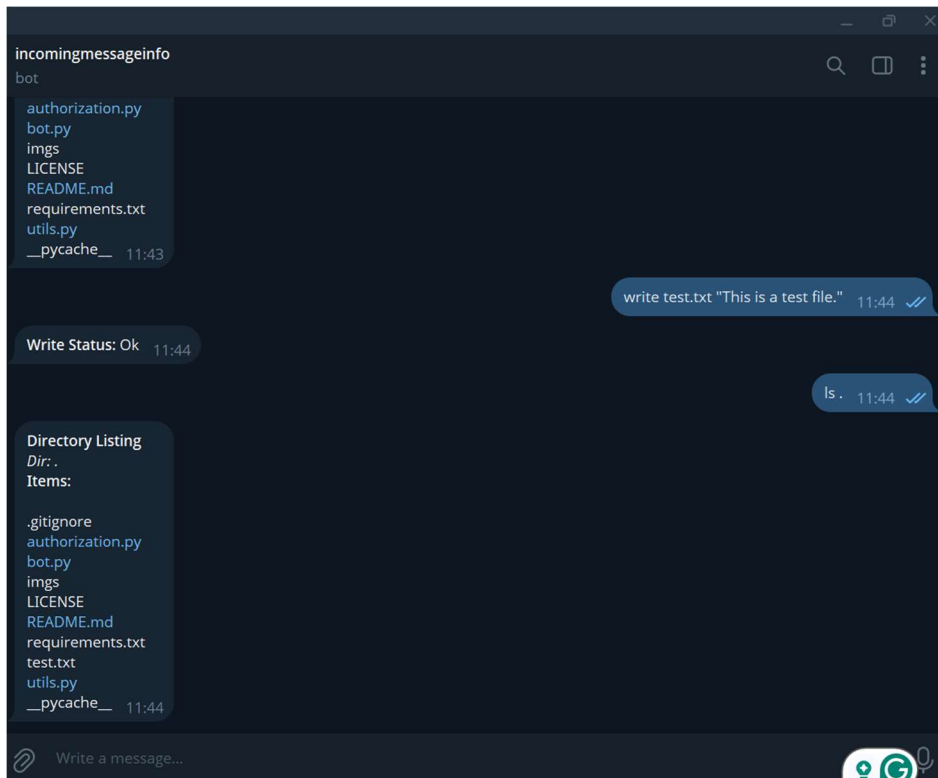
The newly implemented features were then tested. The exec command was verified by sending exec whoami, which correctly returned the username of the account running the script. The download command was tested with download test.txt, and the bot successfully uploaded the newly created text file to the Telegram chat.
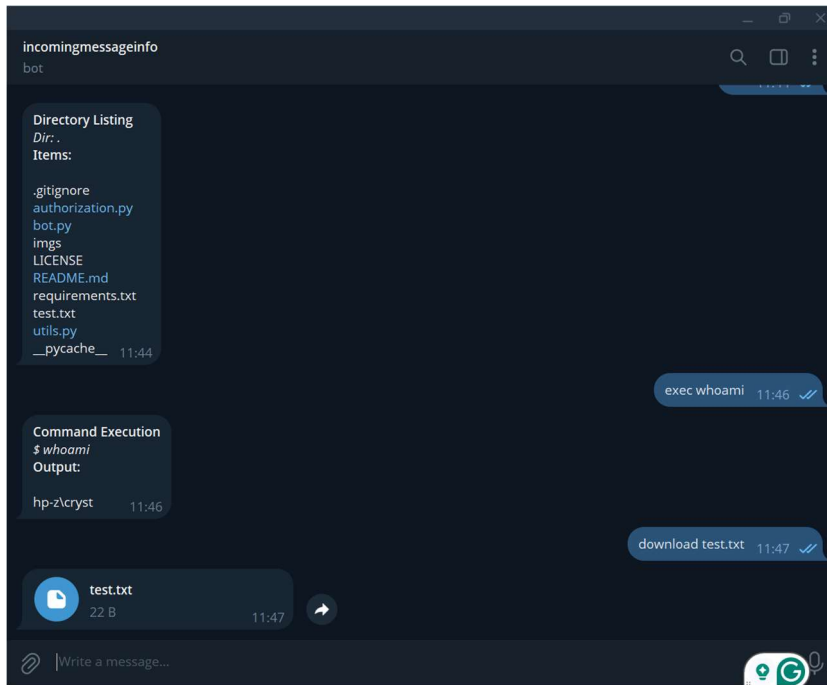
Finally, error handling and termination were checked. A command to list a non-existent directory, ls /nonexistent-directory, was sent. The bot handled the error gracefully and responded with an Err: No such directory. message without crashing. The terminate command was sent, which prompted the bot to reply "Bot shutting down." and cleanly exit the script running in the terminal.

Submitted By: Simran Gupta