# CTF Challenge Documentation

***Created by Yehuda Gurovich***

This document provides an overview of the Capture the Flag (CTF) challenge created for the final assignment of the course. The challenge is designed to test participants' knowledge of various cybersecurity concepts, including HTTP servers, MITM attacks, Wireshark, cryptography, and more. The challenge consists of four stages, each requiring participants to complete specific tasks to progress to the next stage.

## Table of Contents



You are a cyber-security expert working for a covert government agency. Recently, one of the agency's top field agents, known only by their codename "Raven," went missing under mysterious circumstances. Before disappearing, Raven was investigating a dangerous criminal organization. Your mission is to follow Raven's digital trail, decode the messages left behind, and uncover who is behind the organization before they strike.

## Challenge Overview

The CTF has many steps, some related to class material and some others. It consists on 4 stages and requires some knowledge on concepts like: HTTP servers, MITM attacks, Wireshark, curl, Cryptography, python, UDP.

## Setup Instructions

The set up is really simple. Just go to ctf.yehudagurovich.com and get started. If something goes wrong, you can always run the server locally by running the `local_http_server.py` file. The server will run on `localhost:8080`. It should be the same as the cloud server.

## Challenges

#### Stage 1: Setup & Start

In this initial stage, participants are introduced to the challenge by visiting the provided website to gather background information.

#### Stage 2: Finding the Secret Route

This stage involves discovering a hidden route on the website and downloading an executable file essential for the next stage.

**Stage 3: Executing the File & MITM Attack**

This stage involves running the executable file, observing network traffic, and performing a Man-in-the-Middle (MITM) attack to capture a transmitted message.

**Stage 4: Decrypting the Message & Final Message**

In this final stage, participants must decrypt the message obtained from the previous stage and decode it to reveal the final message.

## Files

Files have two categories, the pyserver files and the normal files. pyserver files are files that are required to run the ctf on the cloud in a personal domain so some of the files could be duplicate names but different content (content pertaining the cloud files) or completely duplicate, just needed for the server to run correctly.

**src/pyserver**

`.gcloudignore:` Like the .gitignore but for Google Cloud `app.yaml:` Configuration file for Google Cloud `Dockefile:` Docker configuration file `generate_htmls.py:` Python script to generate the html for different parts of the cloud website `messages.json:` JSON file with the string html parameters for the cloud website `my_http_server.py:` Python script to run the server. It is a simple HTTP server that serves the website using *sockets*, *threading* and the *HTTP protocol*. It is similar to the `local_http_server.py` but with some modifications to run on the cloud. `packets.exe:` Executable file that is downloaded in stage 2. It is a simple client and server that sends a message from the server to the client. It is used to do the *MITM attack* and capture the packets in Wireshark. `parameters.json:` JSON file with the general parameters for the cloud website `requirements.txt:` Python requirements for the cloud server to install the necessary libraries `styles.json:` JSON file with the string css parameters for the cloud website `utils.py:` Python script with some utility functions for the cloud server. Mainly to read the JSON files `victor_blackwood.jpg:` Image for the cloud website

**src/**

`__init__:` Makes the folder a package `decrypt.py:` Script to decrypt the message in stage 4 to test that the encryption and decryption works `encryption.py:` Creates the encrypted message for stage 3 using *Columnar Transposition Cipher* `local_generate_htmls.py:` Python script to generate the html for different parts of the local website `local_http_server.py:` Python script to run the server. It is a simple HTTP server that serves the website using *sockets*, *threading* and the *HTTP protocol*. It is similar to the `my_http_server.py` but with some modifications to run on localhost. `messages.json:` JSON file with the string html parameters for the local website. Has also the messages needed in `mitm_packets.py` `mitm_packets.py:` Python script to generate the packets for the *MITM attack* in stage 3. It generates the packets and sends them to the server and client. It is used to generate `packets.exe` using *pyinstaller* `mitm_packets.spec:` Configuration file for *pyinstaller* to generate the executable `packets.exe:` Executable file that is downloaded in stage 2. It is a simple client and server that sends a message from the server to the client. It is used to do the *MITM attack* and capture the packets in Wireshark. `parameters.json:` JSON file with the general parameters for the local website `styles.json`: JSON file with the string css parameters for the local website. Same as the cloud website `utils.py`: Python script with some utility functions for the local server. Mainly to read the JSON files `victor_blackwood.jpg`: Image for the local website

## Noteworthy code

`my_http_server.py`

```
# Initialize Cloud Logging
client = cloud_logging.Client()
client.setup_logging()

# Setup Python logging
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)


def handle_client(client_socket: socket, client_address: tuple) -> None:
    """
    Handle a client connection for the local HTTP server
    """
    logger.info(f"New connection from {client_address}")

    try:
        # Receive the request data
        request_data = client_socket.recv(
            PARAMETERS["socket_recv_size"]).decode('utf-8')
        logger.info(f"Received request:\n{request_data}")

        # Parse the request
        request_lines = request_data.split('\r\n')
        request_line = request_lines[0]
        parts = request_line.split()
        if len(parts) != 3:
            # Handle the error
            logger.error(f"Invalid request line: {request_line}")
            return
        method, path, _ = parts

        # Parse headers
        headers = {}
        for line in request_lines[1:]:
            if ': ' in line:
                header, value = line.split(': ', 1)
                headers[header] = value

        # Check for User-Agent
        # Google Cloud requires 'user-agent' instead of 'User-Agent'
        user_agent = headers.get('user-agent', '')
        logger.info(f"User-Agent: {user_agent}")
        # Prepare the response
        if path == '/':
```

```
            logger.info("Serving home page")
            status = "200 OK"
            content_type = "text/html"
            content_disposition = ""
            response_body = generate_home_page()
            logger.info(f"User-Agent: {user_agent}")
            # Check if the request is from a command-line or script-based environment
            if any(term in user_agent.lower() for term in ["curl", "wget", "powershell", "python-requests"]):
                logger.info("Adding secret field")
                curl_secret = add_secret_field()
                response_body += f"\n{curl_secret}"

        elif path == '/secretmission':
            logger.info("Serving secret mission page")
            status = "200 OK"
            content_type = "text/html"
            content_disposition = ""
            response_body = generate_secret_mission_page()

        elif path == '/secretfile':
            file_path = "packets.exe"
            with open(file_path, 'rb') as file:
                file_content = file.read()
            response_body = file_content
            status = "200 OK"
            content_type = "application/octet-stream"
            content_disposition = f"attachment; filename={os.path.basename(file_path)}"

        elif path == '/finalmessage':
            logger.info("Serving final message page")
            status = "200 OK"
            content_type = "text/html"
            content_disposition = ""
            response_body = generate_final_message_page()

        elif path == '/victor_blackwood.jpg':
            logger.info("Serving Victor Blackwood image")
            try:
                with open('victor_blackwood.jpg', 'rb') as file:
                    response_body = file.read()
                status = "200 OK"
                content_type = "image/jpeg"
                content_disposition = ""
            except FileNotFoundError:
                logger.error("Image file not found")
                response_body = b"<html><body><h1>404 Not Found</h1></body></html>"
                status = "404 Not Found"
                content_type = "text/html"
                content_disposition = ""
```

```
        else:
            response_body = "<html><body><h1>404 Not Found</h1></body></html>"
            logger.warning(f"Page not found: {path}")
            status = "404 Not Found"
            content_type = "text/html"
            content_disposition = ""

        # Build the HTTP response
        response_header = (
            f"HTTP/1.1 {status}\r\n"
            f"Content-Type: {content_type}\r\n"
            f"Content-Length: {len(response_body)}\r\n"
            "Connection: close\r\n"
        )
        if content_disposition:
            response_header += f"Content-Disposition: {content_disposition}\r\n"
        response_header += "\r\n"

        # Combine header and body
        if isinstance(response_body, bytes):
            response = response_header.encode('utf-8') + response_body
        else:
            response = response_header.encode(
                'utf-8') + response_body.encode('utf-8')

        # Send the response
        client_socket.send(response)

    except Exception as e:
        logger.exception(f"Error handling request from {client_address}: {e}")

    finally:
        # Close the connection
        client_socket.close()
        logger.info(f"Connection closed for {client_address}")


def start_server(host: str, port: str) -> None:
    """
    Start and handle the local HTTP server
    """
    server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    server_socket.bind((host, port))
    server_socket.listen(PARAMETERS["listen_time"])

    logger.info(f"Server listening on {host}:{port}")

    while True:
        client_sock, client_address = server_socket.accept()
        client_handler = threading.Thread(
            target=handle_client, args=(client_sock, client_address))
        client_handler.start()
```

`encryption.py`

```python
def generate_column_cipher_encryption(key: str, message: str) -> str:
    """
    Generate a columnar cipher_encryption from a key and a message.
    """
    # Pad the message
    key_length = len(key)
    padded_message = message.replace(" ", "*")
    padding_length = (key_length - len(padded_message) %
                      key_length) % key_length
    padded_message += "*" * padding_length

    # Create the matrix
    matrix = [padded_message[i:i+key_length]
              for i in range(0, len(padded_message), key_length)]

    # Create sorted column indices
    column_indices = sorted(range(key_length), key=lambda k: key[k])

    # Reorder columns and join
    encrypted_message = ''.join(
        ''.join(row[i] for row in matrix) for i in column_indices)

    return encrypted_message
```

`decrypt.py`

This file is not required for the CTF, but it contains the decryption function to show that the encryption and decryption work. While on the CTF, the decryption is done by creating a script that decrypts the message like the one below.

```python
def decrypt_column_cipher(key: str, encrypted_message: str) -> str:
    """
    Decrypt a message that was encrypted using the columnar cipher.
    """
    # Calculate the number of rows
    key_length = len(key)
    message_length = len(encrypted_message)
    num_rows = (message_length + key_length - 1) // key_length

    # Create sorted column indices (same as in encryption)
    column_indices = sorted(range(key_length), key=lambda k: key[k])

    # Create empty matrix
    matrix = [[''] * key_length for _ in range(num_rows)]

    # Fill the matrix column by column
```

```
    char_index = 0
    for col in column_indices:
        for row in range(num_rows):
            matrix[row][col] = encrypted_message[char_index]
            char_index += 1

    # Read the matrix row by row
    decrypted_message = ''.join(''.join(row) for row in matrix)

    # Remove padding
    return decrypted_message.rstrip('*').replace('*', ' ')


def reorganize_packets(packets: str, split_length: int) -> str:
    """
    Reorganize the packets to extract the hidden message. Make sure packets is
formatted correctly.
    """
    packets = packets.split('.')
    # Extract the packets that contain "ctf"
    packets = [re.sub(r'(ctf\d{2}).*', r'\1', packet)
               for packet in packets if 'ctf' in packet]
    # Obtain the last split_length characters of each packet
    packets = [packet[-split_length:] for packet in packets]
    # Sort the packets by the integer value of the last two characters
    packets = sorted(packets, key=lambda packet: int(packet[-2:]))
    hidden_message = ''.join(packet[:-5] for packet in packets)
    return b64decode(hidden_message).decode().rstrip('\x00')
```

`mitm_packets.py`

**VERY NOTEWORTHY FILE** This file becomes an executable that is downloaded in stage 2. It is used to generate the packets for the MITM attack in stage 3. It generates the packets and sends them from the server to the client. It is used to generate `packets.exe` using *pyinstaller* and `mitm_packets.spec`.

```
# Event to synchronize server and client
ready_to_receive = threading.Event()

# Load messages and parameters
MESSAGES = open_json_file("messages.json")
PARAMETERS = open_json_file("parameters.json")

SPLIT_LENGTH = PARAMETERS["split_length"]
TOTAL_NUMBER_OF_PACKETS = PARAMETERS["total_number_of_packets"]
SERVER_IP = PARAMETERS["server_ip"]
SERVER_PORT = PARAMETERS["server_port"]
CLIENT_PORT = PARAMETERS["client_port"]
BUFFER_SIZE = PARAMETERS["buffer_size"]
TIMEOUT = PARAMETERS["timeout"]
MAX_WAIT_TIME = PARAMETERS["max_wait_time"]
```

```
def create_message_parts() -> List[str]:
    """
    Creates the message parts to be sent in the packets
    """
    column_cipher_message = generate_column_cipher_encryption(
        PARAMETERS["key"], MESSAGES["column_cipher_message"]).encode()

    padding_length = (SPLIT_LENGTH - (len(column_cipher_message) %
                      SPLIT_LENGTH)) % SPLIT_LENGTH
    padded_message = column_cipher_message + b'\x00' * padding_length

    max_index_length = len(str(len(padded_message) // SPLIT_LENGTH))

    return [
        b64encode(padded_message[i:i + SPLIT_LENGTH]).decode() +
        f"ctf{i // SPLIT_LENGTH:0{max_index_length}d}"
        for i in range(0, len(padded_message), SPLIT_LENGTH)
    ]


def create_fake_message(length: int, max_index: int, max_index_length: int) ->
str:
    '''
    Creates fake messages with ftc ending
    '''
    base_fake_message = ''.join(random.choices(
        string.ascii_letters + string.digits, k=length))
    return f"{base_fake_message[:length - 5]}ftc{random.randint(0,
max_index):0{max_index_length}d}"


def create_packets() -> List[Packet]:
    '''
    Creates real and fake packets, shuffles them and adds the clue message at the
beginning
    '''
    message_parts = create_message_parts()
    message_length = len(message_parts[0])
    max_index = len(message_parts)
    max_index_length = len(str(max_index))

    packets = []

    # Create packets with real message parts
    for part in message_parts:
        packet = Ether(src=RandMAC(), dst=RandMAC()) / \
            IP(src=RandIP(), dst=RandIP()) / \
            UDP(sport=RandShort(), dport=RandShort()) / \
            Raw(part.encode())
        packets.append(packet)

    # Create packets with fake messages
    for _ in range(len(message_parts), TOTAL_NUMBER_OF_PACKETS - 1):
```

```
        fake_message = create_fake_message(
            message_length, max_index, max_index_length)
        packet = Ether(src=RandMAC(), dst=RandMAC()) / \
            IP(src=RandIP(), dst=RandIP()) / \
            UDP(sport=RandShort(), dport=RandShort()) / \
            Raw(fake_message.encode())
        packets.append(packet)

    random.shuffle(packets)

    # Insert the MITM message at the beginning
    mitm_packet = Ether(src=RandMAC(), dst=RandMAC()) / \
        IP(src=RandIP(), dst=RandIP()) / \
        UDP(sport=RandShort(), dport=RandShort()) / \
        Raw(MESSAGES["MITM_message"].encode())
    packets.insert(0, mitm_packet)

    return packets


def start_server():
    """
    Starts the server and sends the packets
    """
    packets = create_packets()
    with socket.socket(socket.AF_INET, socket.SOCK_DGRAM) as udp_socket:
        udp_socket.bind((SERVER_IP, SERVER_PORT))
        print("Server is waiting for the client to be ready...")
        ready_to_receive.wait()
        print("Server is sending packets...")
        client_address = (SERVER_IP, CLIENT_PORT)
        for packet in packets:
            udp_socket.sendto(bytes(packet), client_address)
            time.sleep(0.5)
        print(f"{len(packets)} packets sent.")


def start_client():
    """
    Starts the client and receives the packets
    """
    time.sleep(1)
    ready_to_receive.set()
    with socket.socket(socket.AF_INET, socket.SOCK_DGRAM) as udp_socket:
        udp_socket.bind((SERVER_IP, CLIENT_PORT))
        print("Client is receiving packets...")
        received_packets = []
        start_time = time.time()
        while time.time() - start_time < MAX_WAIT_TIME and len(received_packets) <
TOTAL_NUMBER_OF_PACKETS:
            try:
                udp_socket.settimeout(TIMEOUT)
                packet_data, _ = udp_socket.recvfrom(BUFFER_SIZE)
                packet = Ether(packet_data)
```

```
                received_packets.append(packet)
            except socket.timeout:
                print("Timeout occurred, continuing...")

    print(f"Received {len(received_packets)} packets.")
    print("All packets received." if len(received_packets) ==
          TOTAL_NUMBER_OF_PACKETS else "SOMETHING WENT WRONG!")


def main():
    server_thread = threading.Thread(target=start_server)
    client_thread = threading.Thread(target=start_client)
    server_thread.start()
    client_thread.start()
    client_thread.join()
    server_thread.join()
    print("Finished.")
```

## Tools and Techniques Used

1. Python
2. HTTP servers
3. Sockets
4. Threading
5. Google Cloud Platform
6. Wireshark
7. MITM attacks
8. Cryptography
9. UDP
10. Scapy
11. Pyinstaller
12. Curl
13. HTML
14. CSS
15. JSON
16. Docker
17. Cloud Logging
18. Self-hosted websites and domains
19. Regular expressions
20. Base64 encoding
21. Simple server-client communication