Mission Name

Fliying to Euphea

History Background

After being in Paris with the librarian his next step is EUPHEA. Claire and Ethan must obtain authorization in the Recon Car to travel to Euphea.

Technical High-Level Overview

A network dump from a computer connected to a system that manages authorizations is provided to the player. This dump contains communication, that simulates traffic with Recon Permissions.

Short Description

Your goal is to analyse a communication from computer connected to Skytech Flight Authorization System and get the new password set up by Flight Authorization System operator.

Mission Description

Your goal is to analyse a communication from computer connected to Skytech Flight Authorization System and get the new password set up by Flight Authorization System operator.

Location

PARIS, FRANCE | PREPARING TO DEPART

Tools

- Wireshark
- https://github.com/GoSecure/pyrdp

Questions

How many computers are involved?

- 3

Which protocol is the most important to solve the challenge?

- RDP

Which is the first password used to enter Skytech Flight Authorization system?

- fla_atz_system

Items

1. Identify the protocol used, keeping in mind ports involved
2. The key to decrypt the traffic is the other file.
3. Use any tool which could analyse packets to get a video based on the PCAP

Write Up

Player must use Wireshark application ir order to identify RDP traffic. To accomplish this, player has to use the file provided with keys.

First of all, player must open evidence:

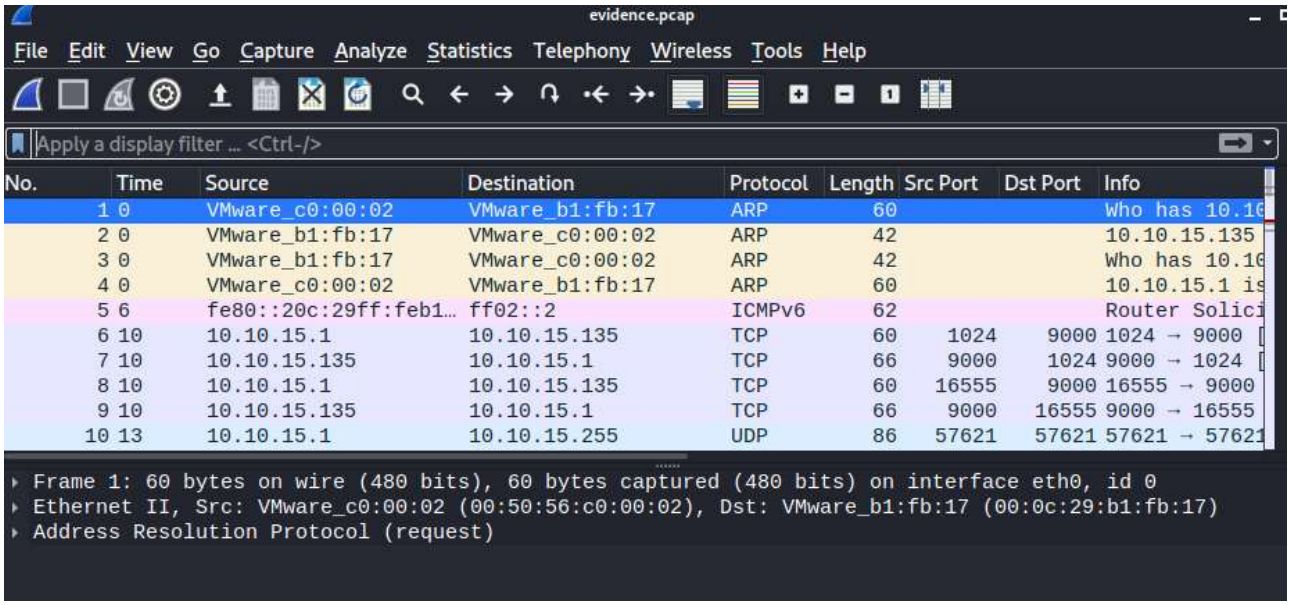

Figure 1

Check protocol Statistics:

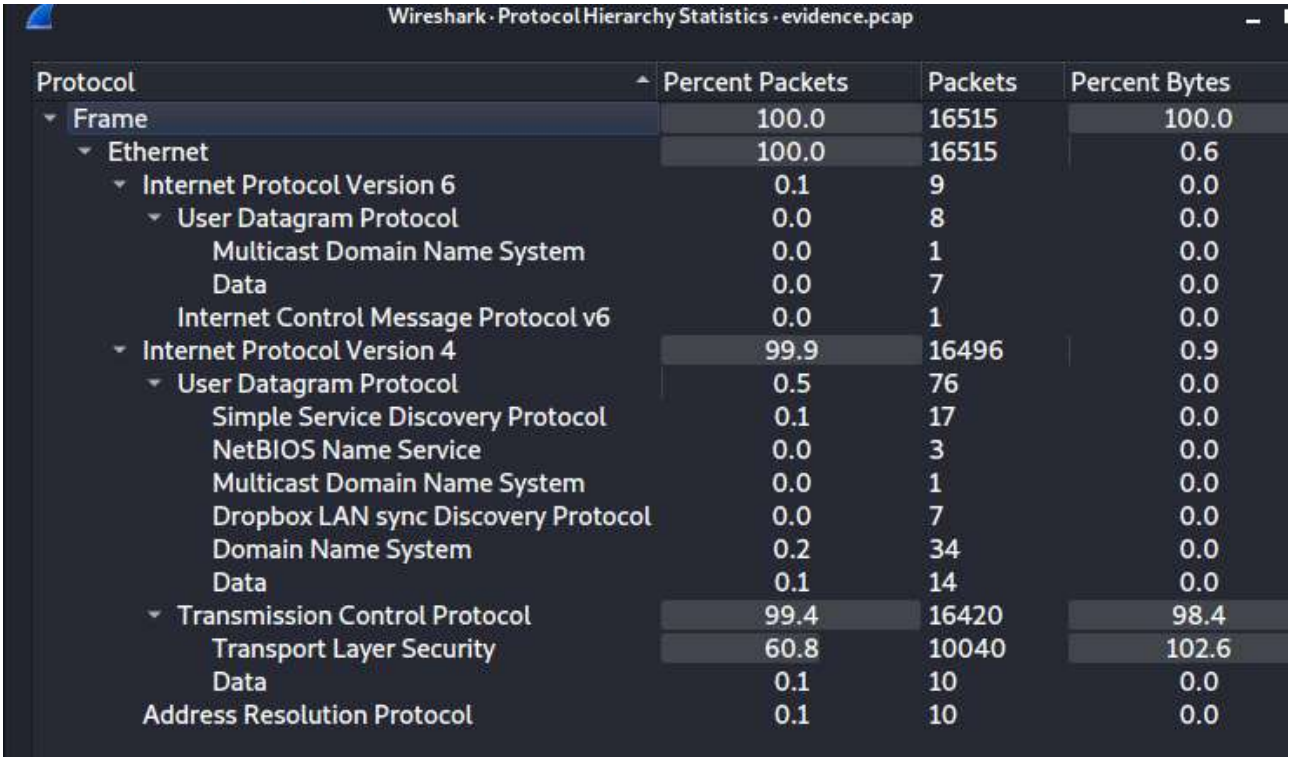
Wireshark · Protocol Hierarchy Statistics · evidence.pcap
Protocol
Percent Packets
Packets
Percent Bytes
Frame 100.0 16515 100.0
Ethernet 100.0 16515 0.6
Internet Protocol Version 6 0.1 9 0.0
User Datagram Protocol 0.0 8 0.0
Multicast Domain Name System 0.0 1 0.0
Data 0.0 7 0.0
Internet Control Message Protocol v6 0.0 1 0.0
Internet Protocol Version 4 99.9 16496 0.9
User Datagram Protocol 0.5 76 0.0
Simple Service Discovery Protocol 0.1 17 0.0
NetBIOS Name Service 0.0 3 0.0
Multicast Domain Name System 0.0 1 0.0
Dropbox LAN sync Discovery Protocol 0.0 7 0.0
Domain Name System 0.2 34 0.0
Data 0.1 14 0.0
Transmission Control Protocol 99.4 16420 98.4
Transport Layer Security 60.8 10040 102.6
Data 0.1 10 0.0
Address Resolution Protocol 0.1 10 0.0

Figure 2

Check Conversations:

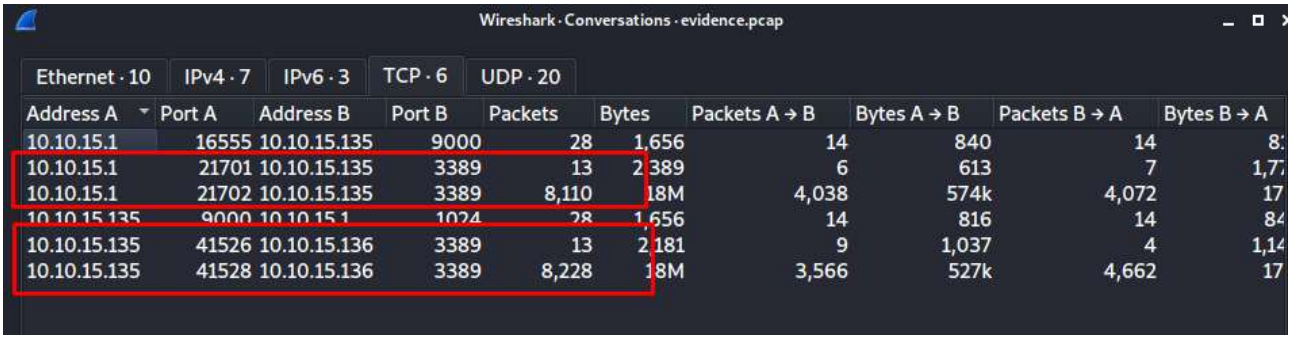

Figure 3

The key is the port shown above: Remote Desktop Protocol.

If we apply RDP filter, there is nothing on Wireshark:

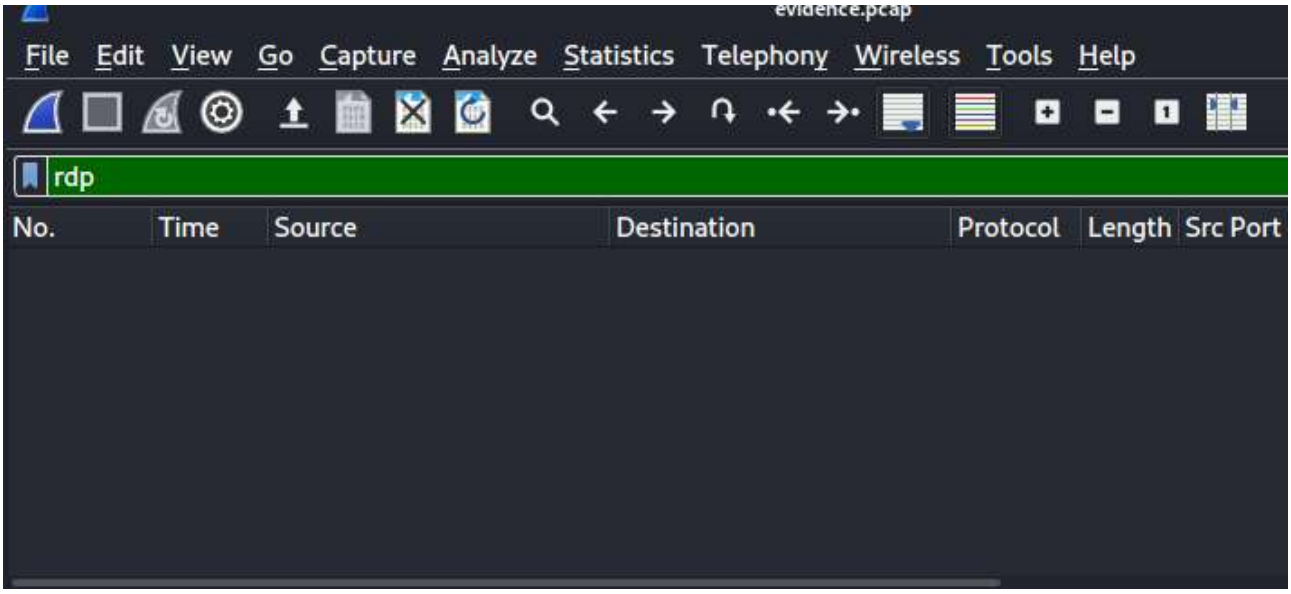

Figure 4

Player has to load TLS keys provided on the file, selecting (Edit -> Preferences -> Protocols -> TLS)

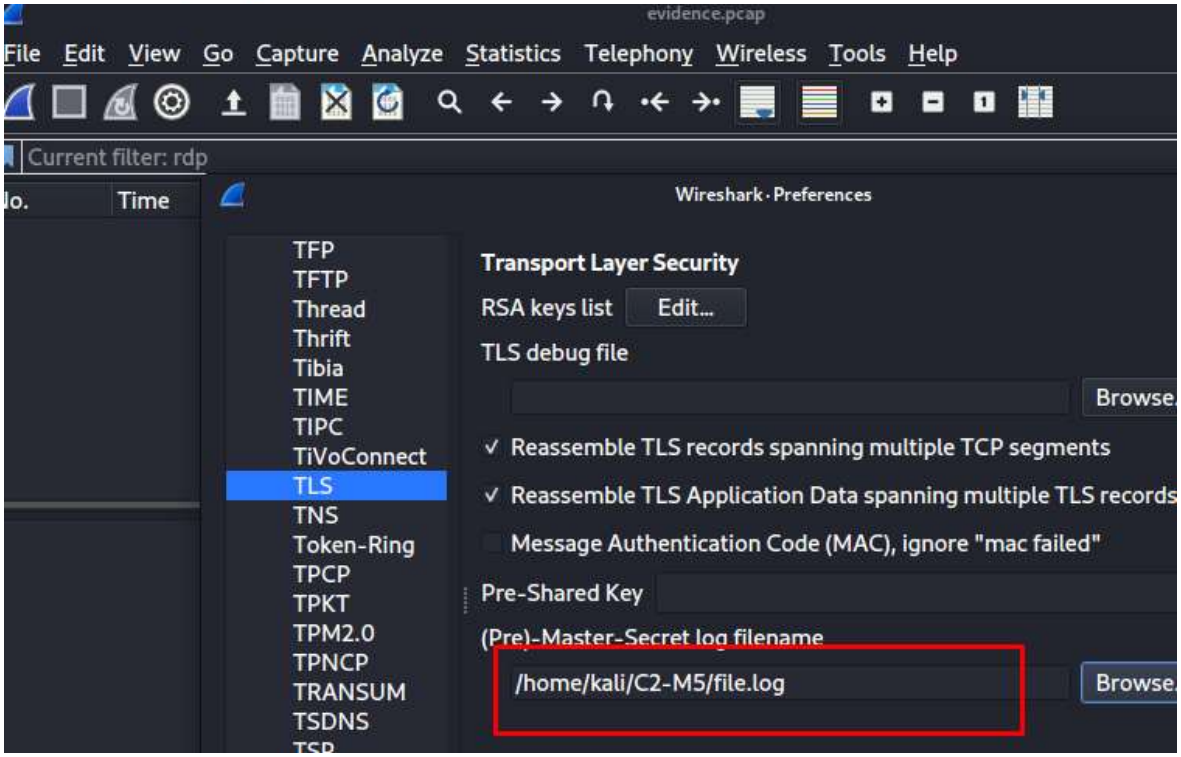

Figure 5

Now, Wiresharks shows RDP protocol:

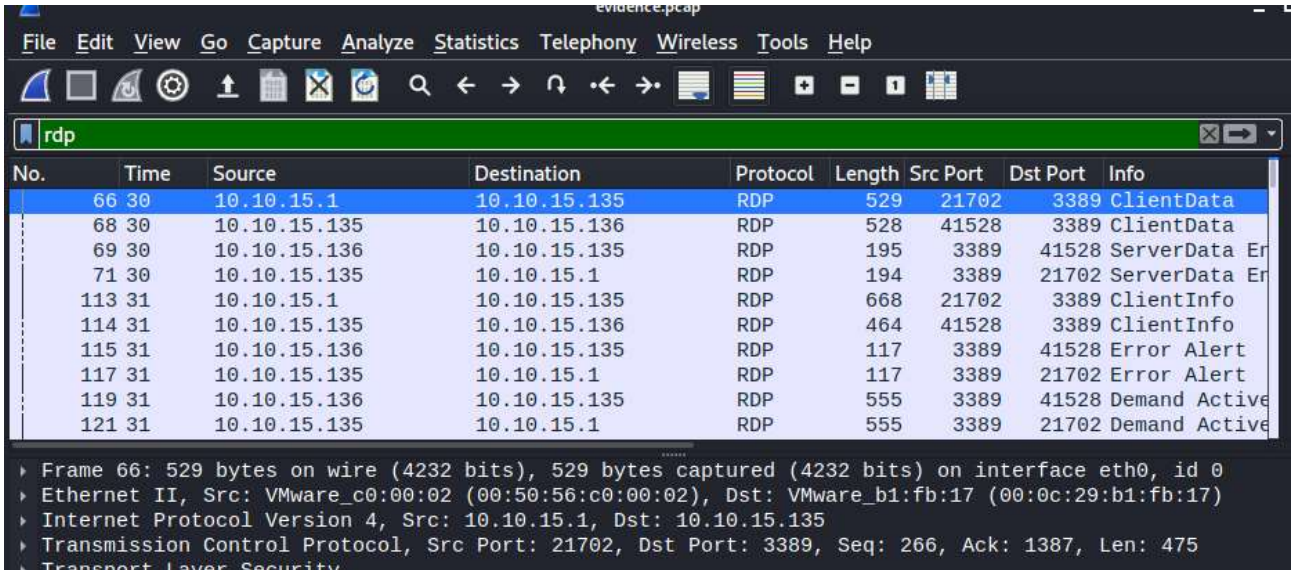

Figure 6

Player has to select the traffic between IP 10.10.15.1 and 10.10.15.135

```
(ip.src == 10.10.15.1 and ip.dst == 10.10.15.135) or (ip.src == 10.10.15.135 and ip.dst == 10.10.15.1)
```

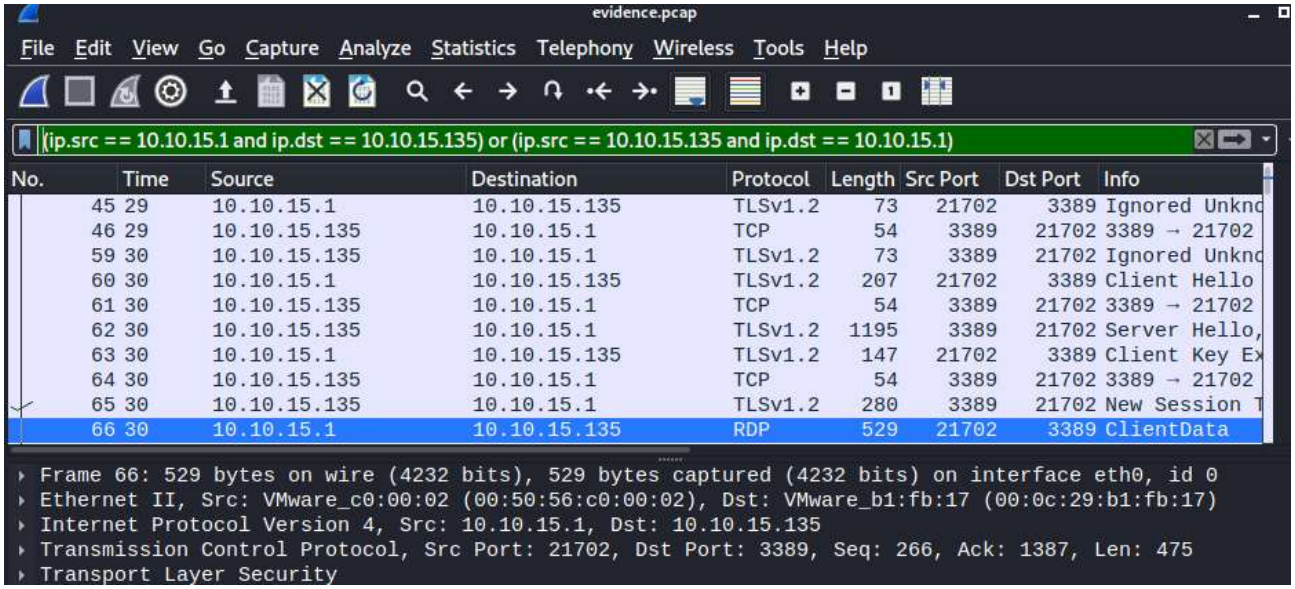

Figure 7

Player has to select Export PDUs to file:

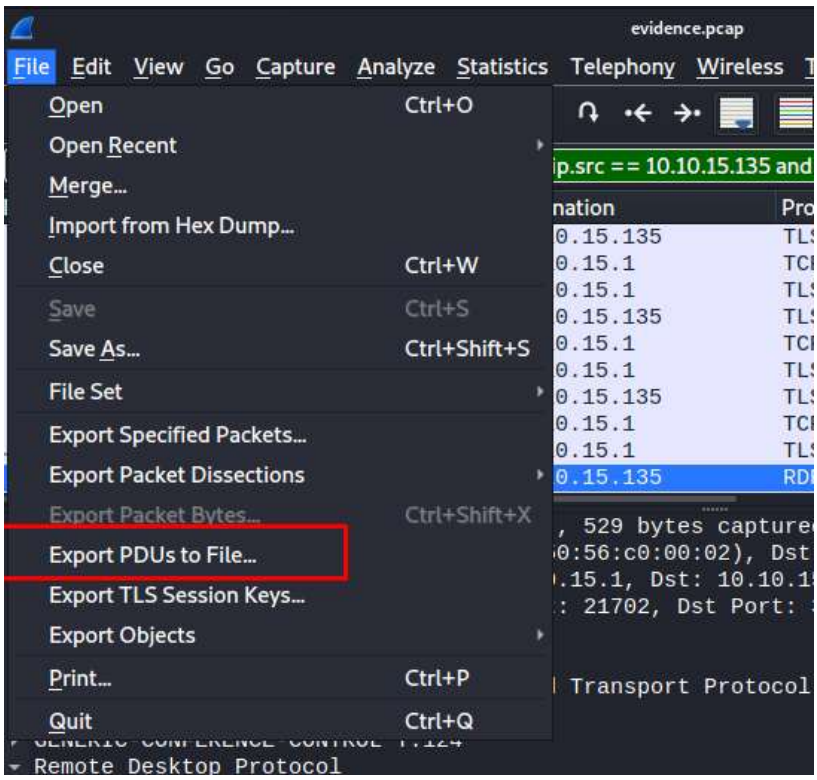

Figure 8

OSI LAYER 7

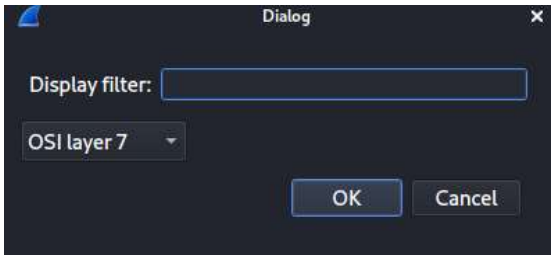

Figure 9

Then Export “Specified Packets”

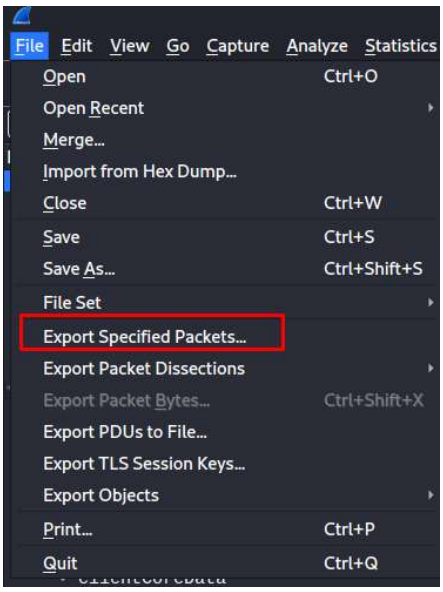

Figure 10

Save output packets: “Export Specified Packets”. It´s very important to save as .PCAP

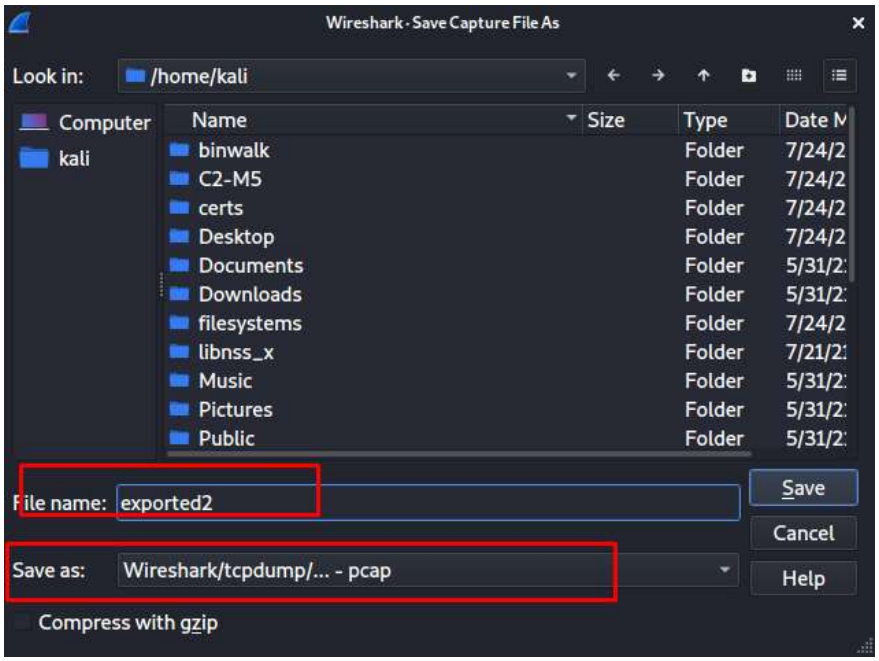

Figure 11

Now we need to install PYRDP:

```
sudo apt install python3 python3-pip python3-dev python3-setuptools python3-venv \
      build-essential python3-dev git openssl \
      libdbus-1-dev libdbus-glib-1-dev \
      notify-osd dbus-x11 libxkbcommon-x11-0 \
      libavformat-dev libavcodec-dev libavdevice-dev \
      libavutil-dev libswscale-dev libswresample-dev libavfilter-dev
```

```
git clone https://github.com/gosecure/pyrdp.git
```

```
cd pyrdp
```

```
python3 -m venv venv
```

```
source /venv/bin/activate
```

```
pip3 install -U pip setuptools wheel
```

```
pip3 install -U -e '.[full]'
```

```
pip3 install ‘r requirements .txt
```

```
sudo apt install libxcb-cursor0
```

Modify as follow:

```
sudo rm ../converter/utils.py
```

```
sudo rm ../converter/PCPAConverter.py
```

```
sudo nano ../converter/utils.py
```

```
#
# This file is part of the PyRDP project.
# Copyright (C) 2021, 2022 GoSecure Inc.
# Licensed under the GPLv3 or later.
#
import enum
from typing import Tuple
```

```
from scapy.layers.inet import IP, TCP
from scapy.layers.l2 import Ether

from pyrdp.convert.JSONEventHandler import JSONEventHandler
from pyrdp.player import HAS_GUI

from pyrdp.convert.pyrdp_scapy import *

"""
Supported conversion handlers.

The class constructor signature must be `__init__(self, output_path: str, progress=None)`
"""
HANDLERS = {"replay": (None, "pyrdp"), "json": (JSONEventHandler, "json")}

if HAS_GUI:
    from pyrdp.convert.MP4EventHandler import MP4EventHandler
    HANDLERS["mp4"] = (MP4EventHandler, "mp4")
else:
    # Class stub for when MP4 support is not available.
    # It would be a good idea to refactor this so that Mp4EventHandler is
    # acquired through some factory object that checks for GUI support
    # once we add more conversion handlers.
    class MP4EventHandler():
        def __init__(self, _unused: str):
            pass


class TCPFlags(enum.IntEnum):
    FIN = 0x01
    SYN = 0x02
    RST = 0x04
    PSH = 0x08
    ACK = 0x10
    URG = 0x20
    ECE = 0x40
    CWR = 0x80


class InetAddress:
    def __init__(self, ip: str, port: int):
        self._ip = ip
        self._port = port
```

```
    @property
    def ip(self) -> str:
        return self._ip

    @property
    def port(self) -> int:
        return self._port

    def __eq__(self, other):
        if isinstance(other, str):  # Verificar si other es una cadena
            return self.ip == other
        elif isinstance(other, InetAddress):
            return self.ip == other.ip and self.port == other.port
        else:
            return False

    def __str__(self):
        return f"{self._ip}:{self._port}"

def extractInetAddressesFromPDUPacket(packet) -> Tuple[InetAddress, InetAddress]:
    """Returns the src and dst InetAddress (IP, port) from a PDU packet"""
    x = ExportedPDU(packet.load)
    return (InetAddress(x.src, x.sport), InetAddress(x.dst, x.dport))


def createHandler(format: str, outputFileBase: str, progress=None) -> Tuple[str, str]:
    """
    Gets the appropriate handler and returns the filename with extension.
    Returns None if the format is replay.
    TODO: Returning None if the format is replay is kind of janky. This could use a refactor to handle replays and other formats differently.
    """

    if format not in HANDLERS:
        print("[-] Unsupported conversion format.")
        sys.exit(1)

    HandlerClass, ext = HANDLERS[format]
    outputFileBase += f".{ext}"
    return HandlerClass(outputFileBase, progress=progress) if HandlerClass else None, outputFileBase
```

```
class ExportedPDU(Packet):
    """60 byte EXPORTED_PDU header."""
    # We could properly parse the EXPORTED_PDU struct, but we are mostly dealing with IP exported PDUs
    # so let's just wing it.
    name = "ExportedPDU"
    fields_desc = [
                IntField("tag1Num", None),  # 4
                StrFixedLenField("proto", None, length=4),  # 8
                IntField("tag2Num", None),  # 12
                IPField("src", None),  # 16
                IntField("tag3Num", None),  # 20
                IPField("dst", None),  # 24
                IntField("tag4Num", None),  # 28
                IntField("portType", None),  # 32
                IntField("tag5Num", None),  # 36
                IntField("sport", None),  # 40
                IntField("tag6Num", None),  # 44
                IntField("dport", None),  # 48
                IntField("tag7Num", None),  # 52
                IntField("frame", None),   # 56
                IntField("endOfTags", None),  # 60
    ]


# noinspection PyUnresolvedReferences
def tcp_both(p) -> str:
    """Session extractor which merges both sides of a TCP channel."""

    if "TCP" in p:
        return str(
            sorted(["TCP", p[IP].src, p[TCP].sport, p[IP].dst, p[TCP].dport], key=str)
        )

    # Need to make sure this is OK when non-TCP, non-exported data is present.
    if Ether not in p:
        x = ExportedPDU(p.load)
        return str(
                sorted([x.proto.upper(), x.src, x.sport, x.dst, x.dport], key=str)
        )

    return "Unsupported"
```

```
# noinspection PyUnresolvedReferences
def findClientRandom(stream: PacketList, limit: int = 20) -> str:
    """Find the client random offset and value of a stream."""
    for n, p in enumerate(stream):
        if n >= limit:
            return ""  # Didn't find client hello.
        try:
            tls = p[TCP].payload
            hello = tls.msg[0]
            if isinstance(hello, TLSClientHello):
                return (pkcs_i2osp(hello.gmt_unix_time, 4) + hello.random_bytes).hex()
        except AttributeError as e:
            continue  # Not a TLS packet.

    return ""


def loadSecrets(filename: str) -> dict:
    secrets = {}
    with open(filename, "r") as f:
        for line in f:
            line = line.strip()
            if line == "" or not line.startswith("CLIENT"):
                continue

            parts = line.split(" ")
            if len(parts) != 3:
                continue

            [t, c, m] = parts

            # Parse the secret accordingly.
            if t == "CLIENT_RANDOM":
                secrets[c] = {"client": bytes.fromhex(c), "master": bytes.fromhex(m)}
    return secrets

def canExtractSessionInfo(session: PacketList) -> bool:
    packet = session[0]
    # TODO: Eventually we should be able to wrap the session as an ExportedSession
    # and check for the presence of exported.
    return IP in packet or Ether not in packet
```

```
def getSessionInfo(session: PacketList) -> Tuple[InetAddress, InetAddress, float, bool]:
    """Attempt to retrieve an (src, dst, ts, isPlaintext) tuple for a data stream."""
    packet = session[0]

    # FIXME: This relies on the fact that decrypted traces are using EXPORTED_PDU and
    #        thus have no `Ether` layer, but it is technically possible to have a true
    #        plaintext capture with very old implementations of RDP.
    if TCP in packet:
        # Assume an encrypted stream...
        return (InetAddress(packet[IP].src, packet[IP][TCP].sport),
                InetAddress(packet[IP].dst, packet[IP][TCP].dport),
                packet.time, False)
    elif Ether not in packet:
        # No Ethernet layer, so assume exported PDUs.
        src, dst = extractInetAddressesFromPDUPacket(packet)
        return (src, dst, packet.time, True)

    raise Exception("Invalid stream type. Must be TCP/TLS or EXPORTED PDU.")
```

sudo nano ../converter/PCPAConverter.py

```
#
# This file is part of the PyRDP project.
# Copyright (C) 2021 GoSecure Inc.
# Licensed under the GPLv3 or later.
#
import math
import traceback
from pathlib import Path
from typing import Dict, List, Tuple

from progressbar import progressbar
from scapy.layers.inet import TCP
from scapy.layers.tls.record import TLS
from pyrdp.convert.pyrdp_scapy import *

from pyrdp.convert.Converter import Converter
from pyrdp.convert.ExportedPDUStream import ExportedPDUStream
from pyrdp.convert.TLSPDUStream import TLSPDUStream
from pyrdp.convert.PCAPStream import PCAPStream
from pyrdp.convert.RDPReplayer import RDPReplayer
```

```
from pyrdp.convert.utils import tcp_both, getSessionInfo, findClientRandom, createHandler, canExtractSessionInfo


class PCAPConverter(Converter):
    SESSIONID_FORMAT = "{timestamp}_{src}-{dst}"

    def __init__(self, inputFile: Path, outputPrefix: str, format: str, secrets: Dict = None, srcFilter = None, dstFilter = None, listOnly = False):
        super().__init__(inputFile, outputPrefix, format)
        self.secrets = secrets if secrets is not None else {}
        self.srcFilter = srcFilter if srcFilter is not None else srcFilter
        self.dstFilter = dstFilter if dstFilter is not None else dstFilter
        self.listOnly = listOnly

    def checkSrcExcluded(self, src: str):
        return self.srcFilter and src not in self.srcFilter

    def checkDstExcluded(self, dst: str):
        return self.dstFilter and dst not in self.dstFilter

    def process(self):
        streams = self.listSessions()

        if self.listOnly:
            return

        exitCode = 0
        for startTimeStamp, stream in streams:
            try:
                self.processStream(startTimeStamp, stream)
            except Exception as e:
                trace = traceback.format_exc()
                print() # newline
                print(trace)
                print(f"[-] Failed: {e}")
                exitCode = 1
        return exitCode

    def listSessions(self) -> List[Tuple[int, PCAPStream]]:
        print(f"[*] Analyzing PCAP '{self.inputFile}' ...")
        bind_layers(TCP, TLS)
        pcap = sniff(offline=str(self.inputFile), session=TCPSession)
```

```
        sessions = pcap.sessions(tcp_both)

        if len(sessions.values()) == 0:
            print("No sessions found!")
            return []

        streams: List[Tuple[int, PCAPStream]] = []

        for session in sessions.values():
            if not canExtractSessionInfo(session):
                # Skip unsupported sessions (e.g: UDP sessions and such)
                continue

            client, server, startTimeStamp, plaintext = getSessionInfo(session)

            if self.checkSrcExcluded(client) or self.checkDstExcluded(server):
                continue

            print(f"    - {client} -> {server} :", end="", flush=True)

            if plaintext:
                print(" plaintext")
                stream = ExportedPDUStream(client, server, session)
            else:
                clientRandom = findClientRandom(session)

                if clientRandom in self.secrets:
                    print(" TLS, master secret available (!)")
                    stream = TLSPDUStream(client, server, session, self.secrets[clientRandom]["master"])
                else:
                    print(" TLS, unknown master secret")
                    continue

            streams.append((startTimeStamp, stream))

        return streams

    def processStream(self, startTimeStamp: int, stream: PCAPStream):
        startTimeStamp = time.strftime("%Y%m%d%H%M%S",
time.gmtime(math.floor(startTimeStamp)))
        sessionID = PCAPConverter.SESSIONID_FORMAT.format(**{
            "timestamp": startTimeStamp,
```

```
            "src": stream.client,
            "dst": stream.server
        })

        handler, _ = createHandler(self.format, self.outputPrefix + sessionID)
        replayer = RDPReplayer(handler, self.outputPrefix, sessionID)

        print(f"[*] Processing {stream.client} -> {stream.server}")

        try:
            for data, timeStamp, src, _dst in progressbar(stream):
                replayer.setTimeStamp(timeStamp)
                replayer.recv(data, src == stream.client)
        except StopIteration:
            # Done processing the stream.
            pass

        try:
            replayer.tcp.recordConnectionClose()
            if handler:
                handler.cleanup()
        except struct.error:
            sys.stderr.write("[!] Couldn't close the session cleanly. Make sure that --src and --dst are
correct.")

        print(f"\n[+] Successfully wrote '{replayer.filename}'")
```

- python3 convert.py --src 10.10.15.1 --dst 10.10.15.135 -o video -f mp4 /home/challenges/Claire/HARD/10_Flying_to_Euphea/hard/evidence/exported2.pcap

```
└─$ python3 convert.py --src 10.10.15.1 --dst 10.10.15.135 -o video -f mp4 /home/challenges/
[*] Analyzing PCAP '/home/challenges/Claire/HARD/10_Flying_to_Euphea/hard/evidence/exported2
    - 10.10.15.1:21702 → 10.10.15.135:3389 : plaintext
[*] Processing 10.10.15.1:21702 → 10.10.15.135:3389
  7% (416 of 5777) |##                          | Elapsed Time: 0:00:21 ETA:   0:04:42
```

Figure 12

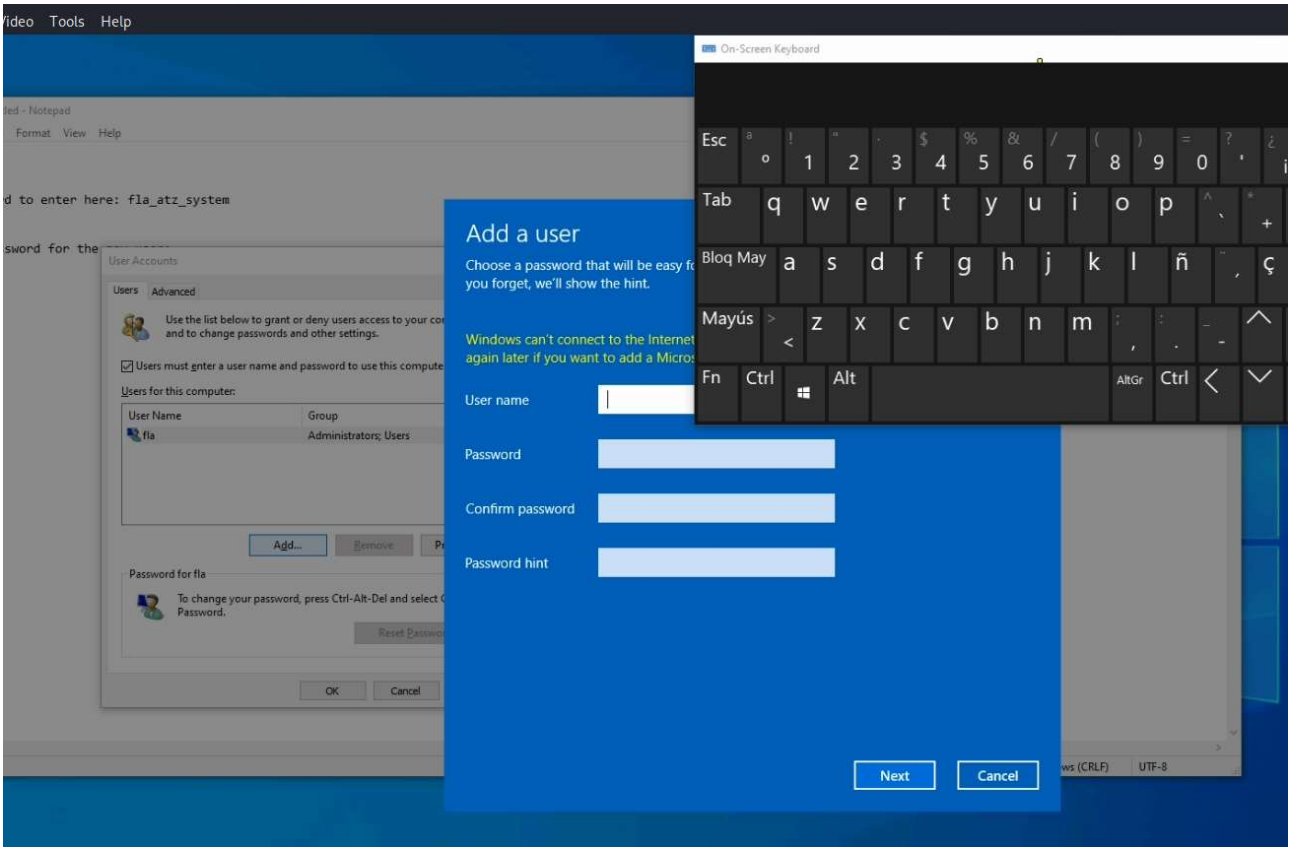

Figure 12

Player has to open the video (video.mp4) created and see when the new user is created. The big challenge is to follow keystrokes and get the new password:

Flag Information

flag{bigbridgesinmanhattan}