Assignment 1: Defeating SkyNet

A.R.P

August 23, 2016

Contents

1 comms.py 2

2 evil.py 6

3 files.py 7

4 helpers.py 9

5 master_sign.py 10

6 master_view.py 11

7 p2p.py 12

1 comms.py

```
import struct
from Crypto.Hash import SHA256
from Crypto.Hash import HMAC
from Crypto import Random
from Crypto.Random import random
from Crypto.Cipher import AES
from lib.helpers import ANSI_X923_pad, ANSI_X923_unpad
from dh import create_dh_key, calculate_dh_secret

# ~ Global variables ~
# We are using SHA256, thefore each HMAC is 64 bytes long (Hexadecimal string
    format)
HMAC_LEN    = 64
 # We declare that the counter is sent as a 7 byte number max counter value is
     therefore ’1000000’
COUNTER_LEN = 7

class StealthConn(object):
  def __init__(self, conn, client=False, server=False, verbose=True):
    self.conn               = conn
    self.client             = client
    self.server             = server
    self.verbose            = verbose
    self.session_counter    = None
    self.DEK                = None  # Stored in byte format
    self.signing_secret     = None  # Stored in byte format

    self.initiate_session()

  def __print_verbose(self, string):
    if self.verbose:
      print(string)

  def __packet_send(self, data):
    """
    Sends data in a ’packet’.
    """
    # Encode the data’s length into an unsigned two byte int (’H’)
    pkt_len = struct.pack(’H’, len(data))
    self.conn.sendall(pkt_len)
    self.conn.sendall(data)

  def __packet_recv(self):
    """
    Recieves a packet from the network.
    Returns data and length of the data.
    """
    pkt_len_packed = self.conn.recv(struct.calcsize(’H’))
    pkt_len = struct.unpack(’H’, pkt_len_packed)[0]

    return self.conn.recv(pkt_len), pkt_len

  def initiate_session(self):
    """
    Initiates session between client and server bots.
    """
    # Perform the initial connection handshake for agreeing on a shared secret
    if self.server or self.client:

      # Calculate diffie-Hellman key pair and send our public key
      my_public_key, my_private_key = create_dh_key()
      self.__packet_send(bytes(str(my_public_key), "ascii"))

      # Receive their public key
      pubKey, key_len = self.__packet_recv()
      their_public_key = int(pubKey)

      # Obtain a shared secret
      shared_hash = calculate_dh_secret(their_public_key, my_private_key)
      shared_hash = bytes(shared_hash, "ascii")

      # Derive the following from the shared secret
      #   (a) Symmetric (16 byte) DEK (Data Encryption Key)
      #   (b) Initial session counter value
      #   (c) Secret used in the signing of messages (HMAC)
      self.DEK             = shared_hash[:16]
      self.session_counter = int(shared_hash[17])
      self.signing_secret  = shared_hash[17:]

  def send(self, data):
    """
    Encrypt and send data over the network.
    """
    # Create the Cipher with the new IV
    iv     = Random.get_random_bytes(AES.block_size)
    cipher = AES.new(self.DEK, AES.MODE_OFB, iv)

    # Create the HMAC = hash( signing_secret + session_counter + plaintext)
    hmac = HMAC.new(self.signing_secret, digestmod=SHA256)
    hmac.update(bytes(str(self.session_counter), "ascii"))
    hmac.update(data)

    # Construct the data to encrypt = session_counter +  HMAC + data
    hmac            = bytes(str(hmac.hexdigest()), "ascii")
    ctr_str         = bytes(str(self.session_counter), "ascii")
    ctr_str         = ANSI_X923_pad(ctr_str, COUNTER_LEN)
    data_to_encrypt = ctr_str + hmac + data

    # Encrypt the message
    data_to_encrypt = ANSI_X923_pad(data_to_encrypt, cipher.block_size)
    encrypted_data  = cipher.encrypt(data_to_encrypt)

    self.__print_verbose("Original␣data:␣{}".format(data))
    self.__print_verbose("Encrypted␣Data:␣{}".format(repr(encrypted_data)))

    # Append the iv to create the packet
    packet = iv + encrypted_data

    # Send the packet
    self.__print_verbose("Sending␣packet␣of␣length␣{}".format(len(packet)))
    self.__packet_send(packet)
    # Increment the session counter, so that the next message will have a new '
        unique' freshness identifier
    self.session_counter += 1

  def recv(self):
    """
    Recieve and decrypt data from the network.
    """
    # Recieve the paket
    packet, pkt_len = self.__packet_recv()

    # Split the packet into iv and encrypted data
    iv             = packet[:AES.block_size]
    encrypted_data = packet[AES.block_size:]

    # Initialize decrypt cipher and decrypt data
    cipher  = AES.new(self.DEK, AES.MODE_OFB, iv)
    message = cipher.decrypt(encrypted_data)
    message = ANSI_X923_unpad(message, cipher.block_size)

    # Split the data into counter, HMAC and plaintext
    recv_counter = ANSI_X923_unpad(message[:COUNTER_LEN], COUNTER_LEN)
    recv_hmac    = message[COUNTER_LEN:(COUNTER_LEN + HMAC_LEN)]
    plaintext    = message[(COUNTER_LEN + HMAC_LEN):]

    # Calculate our own hmac to verify integrity
    calc_hmac = HMAC.new(self.signing_secret, digestmod=SHA256)
    calc_hmac.update(recv_counter)
    calc_hmac.update(plaintext)

    # Convert to byte format for comparing data
    calc_hmac = bytes(str(calc_hmac.hexdigest()), "ascii")
    this_counter = bytes(str(self.session_counter), "ascii")

    # Perform Anti-Replay check
    if recv_counter == this_counter:
      # Autenticate with HMAC
      if calc_hmac == recv_hmac:
        data = plaintext
        self.__print_verbose("Receiving␣packet␣of␣length␣{}".format(pkt_len))
        self.__print_verbose("Encrypted␣Data:␣{}".format(repr(encrypted_data)))
        self.__print_verbose("Original␣data:␣{}".format(data))

        self.session_counter += 1 # Increment session counter, to keep lock-step
             with the other bot
      else:
        data = None
        self.__print_verbose("Integrity␣Check␣Failed!")
        self.close()
    else:
      data = None
      self.__print_verbose("Replay␣Attack␣detected!")
      self.close()

    return data

  def close(self):
```

```
self.conn.close()
```

2 evil.py

```
# We're using Python's builtin random
# NOTE: This is not cryptographically strong
import random
import time

from lib.helpers import generate_random_string

def bitcoin_mine():
  frames = "\\|/-"
  for i in range(8):
          print("\r%c" % frames[i % len(frames)], end="")
          time.sleep(0.1)

  print()
  # Bitcoin addresses start with a 3 or 1
  return random.choice("13") + generate_random_string(length=30)

def harvest_user_pass():
  names = "Bob␣Tim␣Ben␣Adam␣Lois␣Julie␣Daniel␣Lucy␣Sam␣Stephen␣Matt␣Luke␣Jenny␣
      Becca".split()
  return random.choice(names), generate_random_string(length=10)
```

3 files.py

```
import os
from Crypto.Hash import SHA256
from Crypto.Signature import PKCS1_v1_5
from Crypto.PublicKey import RSA
from Crypto.Cipher import PKCS1_OAEP
from Crypto import Random
from Crypto.Cipher import AES
from lib.helpers import ANSI_X923_pad, ANSI_X923_unpad

# Instead of storing files on disk,
# we'll save them in memory for simplicity
filestore = {}
valuables = []  # Valuable data to be sent to the botmaster
SIGN_LEN = 256  # The length of the signature

pubkey_txt = """-----BEGIN PUBLIC KEY-----
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAw/wc039pKTQ+ArqB0oVE
ouQdNW2XJxE0iTaKggwABqQM01ux4HxJ1obSx2WRI+1XmytQiGEvUp0vSX4sP9W3
gE6eiPtt7S77XRv3xkvL2UfVpoqwq9zrKRupCiSm0XzZodf1WPResWJ/0x9CIFCy
N0b7UprQWz14mCNh+2+GnMfx1kAKabhMMeviuHqkeAlc34hvluQwb6ipa7lrmZnA
/nbRlaflP0esIcjh/rzT0gGMNwrVV66W/aufzntjdQ8sy4EhowL4nG5LJ9cwYNTs
RRlfyjLmVzM06VIs0GvwITT8C8m1NeN69YcA78dwpUc00/ddQNbijbnws1D0bcI7
CwIDAQAB
-----END PUBLIC KEY-----"""


def save_valuable(data):
  valuables.append(data)

def encrypt_for_master(data):
  # Encrypt the file so it can only be read by the bot master
  # Generate a random iv and Key to create AES cipher
  iv      = Random.get_random_bytes(AES.block_size)
  symmkey = Random.get_random_bytes(AES.block_size)
  cipher  = AES.new(symmkey, AES.MODE_OFB, iv)

  # Encrypt the data using the derived symmetric key
  data_to_encrypt = ANSI_X923_pad(data, cipher.block_size)
  ciphertext      = cipher.encrypt(data_to_encrypt)

  # Use the public rsa key to encrypt iv and the symmetric key
  pubkey        = RSA.importKey(pubkey_txt)
  rsa_cipher    = PKCS1_OAEP.new(pubkey)
  encrypt_cipher_info = rsa_cipher.encrypt(iv + symmkey)

  return encrypt_cipher_info + ciphertext

def upload_valuables_to_pastebot(fn):
  # Encrypt the valuables so only the bot master can read them
  valuable_data = "\n".join(valuables)
  valuable_data = bytes(valuable_data, "ascii")
  encrypted_master = encrypt_for_master(valuable_data)

  # "Upload" it to pastebot (i.e. save in pastebot folder)
  f = open(os.path.join("pastebot.net", fn), "wb")
  f.write(encrypted_master)
  f.close()

  print("Saved␣valuables␣to␣pastebot.net/%s␣for␣the␣botnet␣master" % fn)

def verify_file(f):
  # Verify the file was sent by the bot master
  signature = f[:SIGN_LEN]
  pubkey    = RSA.importKey(pubkey_txt)
  h = SHA256.new(f[SIGN_LEN:])
  verifier = PKCS1_v1_5.new(pubkey)

  return verifier.verify(h, signature)

def process_file(fn, f):
  if verify_file(f):
    # If it was, store it unmodified
    # (so it can be sent to other bots)
    # Decrypt and run the file
    filestore[fn] = f
    print("Stored␣the␣received␣file␣as␣%s" % fn)
  else:
    print("The␣file␣has␣not␣been␣signed␣by␣the␣botnet␣master")

def download_from_pastebot(fn):
  # "Download" the file from pastebot.net
  # (i.e. pretend we are and grab it from disk)
  # Open the file as bytes and load into memory
  if not os.path.exists(os.path.join("pastebot.net", fn)):
    print("The␣given␣file␣doesn't␣exist␣on␣pastebot.net")
    return
  f = open(os.path.join("pastebot.net", fn), "rb").read()
  process_file(fn, f)

def p2p_download_file(sconn):
  # Download the file from the other bot
  fn = str(sconn.recv(), "ascii")
  f = sconn.recv()
  print("Receiving␣%s␣via␣P2P" % fn)
  process_file(fn, f)

def p2p_upload_file(sconn, fn):
  # Grab the file and upload it to the other bot
  # You don't need to encrypt it only files signed
  # by the botnet master should be accepted
  # (and your bot shouldn't be able to sign like that!)
  if fn not in filestore:
    print("That␣file␣doesn't␣exist␣in␣the␣botnet's␣filestore")
    return
  print("Sending␣%s␣via␣P2P" % fn)
  sconn.send(bytes(fn, "ascii"))
  sconn.send(filestore[fn])

def run_file(f):
  # If the file can be run,
  # run the commands
  pass
```

4 helpers.py

```python
# We're using Python's builtin random
# NOTE: This is not cryptographically strong
import random
import string

def read_hex(data):
  # Remove any spaces or newlines
  data = data.replace("␣", "").replace("\n", "")
  # Read the value as an integer from base 16 (hex)
  return int(data, 16)

def generate_random_string(alphabet=None, length=8, exact=False):
  if not alphabet:
      alphabet = string.ascii_letters + string.digits
  """
  The line below is called a list comprehension and is the same as:
  letters = []
  for i in range(length):
      # Select a random letter from the alphabet and add it to letters
      letters.append(random.choice(alphabet))
  # Join the letters together with no separator
  return ''.join(letters)
  """
  if not exact:
      length = random.randint(length-4 if length-4 > 0 else 1,length+4)
  return ''.join(random.choice(alphabet) for x in range(length))

# ANSI X.923 pads the message with zeroes
# The last byte is the number of zeroes added
# This should be checked on unpadding
def ANSI_X923_pad(m, pad_length):
  # Work out how many bytes need to be added
  required_padding = pad_length - (len(m) % pad_length)
  # Use a bytearray so we can add to the end of m
  b = bytearray(m)
  # Then k-1 zero bytes, where k is the required padding
  b.extend(bytes("\x00" * (required_padding-1), "ascii"))
  # And finally adding the number of padding bytes added
  b.append(required_padding)
  return bytes(b)

def ANSI_X923_unpad(m, pad_length):
  # The last byte should represent the number of padding bytes added
  required_padding = m[-1]
  # Ensure that there are required_padding - 1 zero bytes
  if m.count(bytes([0]), -required_padding, -1) == required_padding - 1:
      return m[:-required_padding]
  else:
      # Raise an exception in the case of an invalid padding
      raise AssertionError("Padding␣was␣invalid")
```

5 master_sign.py

```
import os
from Crypto.PublicKey import RSA
from Crypto.Hash import SHA256
from Crypto.Signature import PKCS1_v1_5

def sign_file(f):
  # Import the master (private) RSA key to sign file
  masterkey = RSA.importKey(open(’masterkey.pem’, ’rb’).read())

  # Create the signature of the hash that will be pre-pended to the message
  h = SHA256.new(f)
  signer = PKCS1_v1_5.new(masterkey)
  signature = signer.sign(h)

  return signature + f


if __name__ == "__main__":
  fn = input("Which␣file␣in␣pastebot.net␣should␣be␣signed?␣")
  if not os.path.exists(os.path.join("pastebot.net", fn)):
    print("The␣given␣file␣doesn’t␣exist␣on␣pastebot.net")
    os.exit(1)
  f = open(os.path.join("pastebot.net", fn), "rb").read()
  signed_f = sign_file(f)
  signed_fn = os.path.join("pastebot.net", fn + ".signed")
  out = open(signed_fn, "wb")
  out.write(signed_f)
  out.close()
  print("Signed␣file␣written␣to", signed_fn)
```

6 master_view.py

```
import os
from Crypto.PublicKey import RSA
from Crypto.Cipher import PKCS1_OAEP
from Crypto.Cipher import AES
from lib.helpers import ANSI_X923_pad, ANSI_X923_unpad

def decrypt_valuables(f):
  # Import the rsa private key and create the rsa decrypt cipher
  masterkey = RSA.importKey(open('masterkey.pem', 'rb').read())
  rsa_cipher = PKCS1_OAEP.new(masterkey)

  # Calculate the length of the encrypted cipher info (in bytes)
  info_len = int((masterkey.size() + 1)/8)

  # Seperate the encrypted cipher info and message
  encrypt_cipher_info = f[:info_len]
  encrypted_message   = f[info_len:]

  # Decrypt iv and the symmetric key
  info    = rsa_cipher.decrypt(encrypt_cipher_info)
  iv      = info[:AES.block_size]
  symmkey = info[AES.block_size:(AES.block_size*2)]

  # Decrypt the message using the found iv and symmetric key
  cipher  = AES.new(symmkey, AES.MODE_OFB, iv)
  message = cipher.decrypt(encrypted_message)
  message = ANSI_X923_unpad(message, cipher.block_size)

  print(message)


if __name__ == "__main__":
  fn = input("Which␣file␣in␣pastebot.net␣does␣the␣botnet␣master␣want␣to␣view?␣")
  if not os.path.exists(os.path.join("pastebot.net", fn)):
    print("The␣given␣file␣doesn't␣exist␣on␣pastebot.net")
    os.exit(1)
  f = open(os.path.join("pastebot.net", fn), "rb").read()
  decrypt_valuables(f)
```

7 p2p.py

```
import socket
import threading

from lib.comms import StealthConn
from lib.files import p2p_download_file

# Keep track of where our server is
# This is primarily so we don’t try to talk to ourselves
server_port = 1337

def find_bot():
  print("Finding␣another␣bot...")
  port = 1337
  conn = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
  while 1:
    if port == server_port:
      # Don’t connect to yourself, silly bot!
      port += 1
    else:
      try:
        print("Found␣bot␣on␣port␣%d" % port)
        conn.connect(("localhost", port))
        sconn = StealthConn(conn, client=True)
        return sconn
      except socket.error:
        print("No␣bot␣was␣listening␣on␣port␣%d" % port)
        port += 1

def echo_server(sconn):
  while 1:
    data = sconn.recv()
    print("ECHOING>", data)
    sconn.send(data)
    if data == b’X’ or data == b’exit’:
      print("Closing␣connection...")
      sconn.close()
      return

def accept_connection(conn):
  try:
    sconn = StealthConn(conn, server=True)
    # The sender is either going to chat to us or send a file
    cmd = sconn.recv()

    if cmd == b’ECHO’:
      echo_server(sconn)
    elif cmd == b’FILE’:
      p2p_download_file(sconn)
  except socket.error:
    print("Connection␣closed␣unexpectedly")

def bot_server():
  global server_port
  # Every bot is both client & server, so needs to listen for
  # connections. This is to allow for peer to peer traffic.
```

```
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
# Real worms use shifting ports but for simplicity, we won't.
# We'll also assume you may run another bot on your computer
# so if something else is using 1337, we'll keep going up.
while True:
  try:
    s.bind(("localhost", server_port))
    print("Listening␣on␣port␣%d" % server_port)
    break
  except socket.error:
    # Someone is already using that port -- let's go up one
    print("Port␣%d␣not␣available" % server_port)
    server_port += 1

s.listen(5)

while True:
  print("Waiting␣for␣connection...")
  conn, address = s.accept()
  print("Accepted␣a␣connection␣from␣%s..." % (address,))
  # Start a new thread per connection
  # We don't need to specify it's a daemon thread as daemon status is
      inherited
  threading.Thread(target=accept_connection, args=(conn,)).start()
```