Geheime Botschaften: Verschlüsselte Nachrichten als Audio übertragen

Nachrichten zu versenden, die Ende-zu-Ende-verschlüsselt sind, gehört mittlerweile zum Standard vieler Nachrichten-Apps wie Signal, Threema und WhatsApp. Aber wie wäre es mit etwas spielerischem Retro-Charme? **In diesem Projekt wandelst du Textnachrichten in Töne um und versendest sie als Audio-Dateien im Anhang einer E-Mail. Der Empfänger kann diese Audio-Dateien dann mit Hilfe eines zuvor zwischen euch vereinbarten Schlüsselworts entschlüsseln und lesen.** Nicht Eingeweihte hören nur eine Abfolge von Tönen, wie in diesem Beispiel:

Zum Einsatz kommen hierbei zwei Python-Scripte – je eines für den Sender und den Empfänger. Als E-Mail-Provider dient Gmail.

Gmail für den Versand der E-Mails einrichten

Zunächst benötigst du einen E-Mail-Provider, den du aus dem Python-Script des Senders ansteuern und für den Versand der E-Mails nutzen kannst. Hier bietet sich Googles Gmail an, da die Einrichtung unkompliziert ist. **Wichtig:** Falls du bereits eine

Mail-Adresse bei Gmail besitzt, richte dir für dieses Projekt trotzdem eine neue ein. So stellst du sicher, dass zum Beispiel ein fehlerhaftes Versenden von vielen E-Mails hintereinander zu einer vorübergehenden Sperrung deines Kontos führt.

Wie du eine E-Mail-Adresse bei Gmail und sie für den Versand aus einem Python-Script einrichtest, erfährst du in diesem Tutorial.

Die benötigten Python-Bibliotheken

Im Folgenden verwendest du die Bibliotheken numpy und scipy. Die numpy-Bibliothek wird später verwendet, um numerische Operationen durchzuführen, die für die Analyse der Frequenzen in den Audiodaten benötigt werden. Die scipy-Bibliothek enthält die Funktion wavfile.read, die verwendet wird, um die erzeugten WAV-Datei einzulesen und die Audiodaten sowie die Abtastrate zu extrahieren. Diese Bibliotheken sind nicht standardmäßig in Python enthalten und müssen daher manuell installiert werden. Um sie zu installieren, verwenden Sie den folgenden Befehl:

```
pip install numpy scipy
```

Das Script für den Sender der verschlüsselten Nachrichten

Wenn du nun eine E-Mail-Adresse bei Gmail eingerichtet und die beiden benötigten Bibliotheken installiert hast, kann es direkt weitergehen mit dem Python-Script für den Sender. Hier der vollständige Code:

```
# Verschlüsselte Nachrichten - Script für den Sender
# Pollux Labs, polluxlabs.net

import numpy as np
from scipy.io.wavfile import write
import smtplib
from email.message import EmailMessage
import socket

# Parameter, die vom Benutzer bearbeitet werden müssen
user_parameters = {
    "sample_rate": 44100,  # Abtastrate (Hz)
    "bit_duration": 0.1,   # Dauer eines Bits (in Sekunden)
    "freq_0": 1000,        # Frequenz für "0" (Hz)
    "freq_1": 2000,        # Frequenz für "1" (Hz)
       "encryption_key":  "geheimer_schluessel",    # Verschlüsselungsschlüssel (beide Parteien müssen denselben Schlüssel verwenden)
     "sender_email": "Absender-Adresse",  # Absender-E-Mail-Adresse
    "sender_password": "App-Passwort",        # App-Passwort für die Absender-E-Mail
    "receiver_email": "Empfänger-Adresse",      # Empfänger-E-Mail-Adresse
    "email_subject": "Betreff",   # Betreff der E-Mail
     "email_body": "Inhalt der E-Mail, z.B. Hier kommt eine verschlüsselte Nachricht für dich.",  # E-Mail-Inhalt
     "wav_filename": "message.wav"            # Name der zu speichernden WAV-Datei
}

# Funktion zur Erstellung eines Tons für ein Bit
def generate_tone(frequency, duration, sample_rate):
    # Erzeugt eine Zeitachse von 0 bis zur angegebenen Dauer mit der entsprechenden Anzahl an Samples
    t = np.linspace(0, duration, int(sample_rate * duration), endpoint=False)
    # Berechnet den Sinuswert für die gegebene Frequenz über die Zeitachse
    return np.sin(2 * np.pi * frequency * t)
```

```
# Nachricht in Binärdaten umwandeln
def text_to_binary(text):
    # Wandelt jeden Buchstaben der Nachricht in eine 8-Bit Binärdarstellung um
    binary_data = ''.join(format(ord(char), '08b') for char in text)
    return binary_data

# Nachricht mit dem Schlüssel verschlüsseln
def encrypt_message(text, key):
    encrypted_message = ""
    for i in range(len(text)):
        encrypted_char = chr(ord(text[i]) ^ ord(key[i % len(key)]))
        encrypted_message += encrypted_char
    return encrypted_message

# Nachricht in modulierte Audiodaten umwandeln
def encode_to_audio(binary_data, bit_duration, sample_rate, freq_0, freq_1):
    # Initialisiert ein leeres Array für die Audiodaten
    audio = np.array([])
    # Iteriert durch jedes Bit der Binärdaten
    for bit in binary_data:
        # Erzeugt einen Ton für "0" oder "1" und fügt ihn an das Audioarray an
        if bit == '0':
            audio = np.append(audio, generate_tone(freq_0, bit_duration, sample_rate))
        else:
            audio = np.append(audio, generate_tone(freq_1, bit_duration, sample_rate))
    return audio

# Funktion zum Versenden der WAV-Datei per E-Mail
def send_email_with_attachment(receiver_email, subject, body, attachment_path):
    # Absender-E-Mail und Passwort
    sender_email = user_parameters["sender_email"]
    sender_password = user_parameters["sender_password"]
```

```
    # Erstellen der E-Mail-Nachricht
    msg = EmailMessage()
    msg['From'] = sender_email
    msg['To'] = receiver_email
    msg['Subject'] = subject
    msg.set_content(body)

    # Anhang hinzufügen (die WAV-Datei)
    with open(attachment_path, 'rb') as attachment:
                msg.add_attachment(attachment.read(),
maintype='audio', subtype='wav', filename=attachment_path)

    try:
        # SMTP-Server einrichten und die E-Mail senden
        with smtplib.SMTP_SSL('smtp.gmail.com', 465) as smtp:
              smtp.login(sender_email, sender_password)  #
Anmelden am SMTP-Server
            smtp.send_message(msg)  # E-Mail senden
    except socket.gaierror:
         print("Fehler: Der SMTP-Server konnte nicht erreicht
werden. Bitte überprüfen Sie die Serveradresse.")
    except smtplib.SMTPAuthenticationError:
        print("Fehler: Authentifizierung fehlgeschlagen. Bitte
überprüfen Sie Ihre E-Mail-Adresse und Ihr Passwort.")
    except Exception as e:
        print(f"Ein unerwarteter Fehler ist aufgetreten: {e}")

# Hauptprogramm
if __name__ == "__main__":
    # Nachricht erstellen
    email_message = "Das Pferd frisst keinen Gurkensalat."
    # Nachricht verschlüsseln
      encrypted_message = encrypt_message(email_message,
user_parameters["encryption_key"])
    # Verschlüsselte Nachricht in Binärdaten umwandeln
    binary_data = text_to_binary(encrypted_message)
     print(f"Binärdaten der Nachricht: {binary_data[:64]}...")
# Zeigt die ersten 64 Bits der Nachricht (für Debugging)
    # Binärdaten in Audiosignale umwandeln
         audio_data = encode_to_audio(binary_data,
user_parameters["bit_duration"],
```

```
user_parameters["sample_rate"],  user_parameters["freq_0"],
user_parameters["freq_1"])
    # WAV-Datei speichern
    wav_filename = user_parameters["wav_filename"]
    # Speichert die Audiodaten als 16-Bit Integer in eine WAV-
Datei
      write(wav_filename, user_parameters["sample_rate"],
(audio_data * 32767).astype(np.int16))
    print(f"WAV-Datei '{wav_filename}' erfolgreich erstellt!")

    # WAV-Datei per E-Mail versenden
send_email_with_attachment(user_parameters["receiver_email"],
user_parameters["email_subject"],
user_parameters["email_body"], wav_filename)
    print(f"WAV-Datei '{wav_filename}' erfolgreich per E-Mail
versendet!")
```

So funktioniert das Script

Das Sender-Script verschlüsselt eine Nachricht, wandelt sie in Binärdaten um und erzeugt daraus eine Audiodatei, die dann per E-Mail verschickt wird:

1. **Variablen, die bearbeitet werden müssen:** Zu Beginn und auch an einer Stelle weiter unten im Code gibt es Einstellungen, die du vorab unbedingt vornehmen musst. Dazu gehören der geheime Schlüssel (den der Empfänger in seinem Script auch hinterlegen muss), die E-Mail-Adressen, das Passwort aus Gmail und natürlich die zu verschlüsselnde Nachricht selbst.

```
"encryption_key": "geheimer_schluessel",  # Schlüssel (beide
Parteien müssen denselben Schlüssel verwenden)
"sender_email": "Absender-Adresse",  # Absender-E-Mail-Adresse
"sender_password": "App-Passwort",        # App-Passwort für
die Absender-E-Mail
"receiver_email": "Empfänger-Adresse",       # Empfänger-E-
```

```
Mail-Adresse
"email_subject": "Betreff",   # Betreff der E-Mail
"email_body": "Inhalt der E-Mail, z.B. Hier kommt eine verschlüsselte Nachricht für dich.",  # E-Mail-Inhalt
"wav_filename": "message.wav"            # Name der zu speichernden WAV-Datei

email_message = "Das Pferd frisst keinen Gurkensalat."  # Die eigentliche Textnachricht, die verschlüsselt und versendet wird
```

2. **Nachricht mit dem Schlüssel verschlüsseln** Die Funktion encrypt_message() verwendet eine einfache XOR-Verschlüsselung, um die Nachricht zu verschlüsseln. Dabei wird jedes Zeichen der Nachricht mit einem Zeichen des Schlüssels kombiniert:

```
def encrypt_message(text, key):
    encrypted_message = ""
    for i in range(len(text)):
         encrypted_char = chr(ord(text[i]) ^ ord(key[i % len(key)]))
        encrypted_message += encrypted_char
    return encrypted_message
```

Diese Methode sorgt dafür, dass sowohl der Sender als auch der Empfänger denselben Schlüssel benötigen, um die Nachricht zu entschlüsseln.

3. **Nachricht in Binärdaten umwandeln** Nachdem die Nachricht verschlüsselt wurde, wird sie in Binärdaten umgewandelt, um sie später als Audio darstellen zu können:

```
def text_to_binary(text):
    binary_data = ''.join(format(ord(char), '08b') for char in text)
    return binary_data
```

Jedes Zeichen der Nachricht wird in seine 8-Bit-

Binärdarstellung konvertiert, sodass die gesamte Nachricht als eine Folge von Nullen und Einsen dargestellt wird.

4. **Erstellen eines Tons für jedes Bit** Die Funktion generate_tone() erstellt einen Sinuston für ein einzelnes Bit (entweder „0“ oder „1“). Diese Töne werden später aneinandergereiht, um die gesamte Nachricht in Audiodaten darzustellen:

```
def generate_tone(frequency, duration, sample_rate):
    t = np.linspace(0, duration, int(sample_rate * duration), endpoint=False)
    return np.sin(2 * np.pi * frequency * t)
```

Hierbei wird entweder eine Frequenz für „0“ oder eine andere für „1“ verwendet, um die Bits der Nachricht zu unterscheiden.

5. **Nachricht in Audiodaten kodieren** Die Funktion encode_to_audio() wandelt die gesamte Binärnachricht in Audiodaten um, indem sie für jedes Bit den entsprechenden Ton erzeugt und diese aneinanderreiht:

```
def encode_to_audio(binary_data, bit_duration, sample_rate, freq_0, freq_1):
    audio = np.array([])
    for bit in binary_data:
        if bit == '0':
            audio = np.append(audio, generate_tone(freq_0, bit_duration, sample_rate))
        else:
            audio = np.append(audio, generate_tone(freq_1, bit_duration, sample_rate))
    return audio
```

Das Ergebnis ist eine Audiodatei, die die verschlüsselte Nachricht repräsentiert.

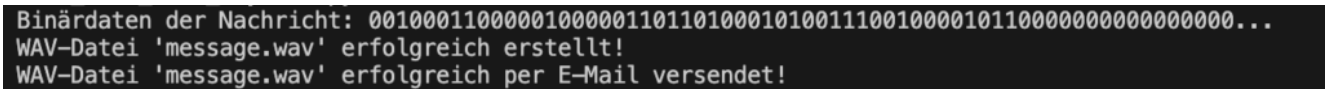
6. **Erstellen und Versenden der Audiodatei** Nachdem die

Audiodaten erstellt wurden, wird die Nachricht als .wav-Datei gespeichert und per E-Mail versendet:

```
write(wav_filename, sample_rate, (audio_data * 32767).astype(np.int16))
```

Diese Zeile speichert die Audiodaten als 16-Bit-Integer-Werte in einer WAV-Datei auf deinem Computer, die dann mit der Funktion send_email_with_attachment() per E-Mail versendet wird.

Ergänze das Script nun um deine Daten und hinterlege die Nachricht, die du versenden möchtest. Lass anschließend das Script einmal laufen – im Terminal solltest du die Nachricht lesen können, dass die E-Mail mit der Audio-Datei an deinen Empfänger gesendet wurde:



Und das war es für den Sender – nun zur anderen Seite, dem Empfänger deiner verschlüsselten Nachricht.

Das Script für den Empfänger

Der Empfänger der Nachricht benötigt also ein eigenes Script, das es ihm ermöglicht, die verschlüsselten Audiodaten wieder in Text umzuwandeln. Hier das vollständige Python-Script:

```
# Verschlüsselte Nachrichten - Script für den Sender
# Pollux Labs, polluxlabs.net

import numpy as np
from scipy.io.wavfile import read

# Parameter, die vom Benutzer bearbeitet werden müssen
user_parameters = {
    "sample_rate": 44100,  # Abtastrate (Hz)
```

```
    "bit_duration": 0.1,   # Dauer eines Bits (in Sekunden)
    "freq_0": 1000,        # Frequenz für "0" (Hz)
    "freq_1": 2000,        # Frequenz für "1" (Hz)
       "encryption_key": "geheimer_schluessel",   # Verschlüsselungsschlüssel (beide Parteien müssen denselben Schlüssel verwenden)
     "wav_filename": "message.wav"          # Name der zu lesenden WAV-Datei
}

# Funktion zum Dekodieren der Audiodaten in Binärdaten
def decode_audio_to_binary(audio_data, bit_duration, sample_rate, freq_0, freq_1):
    bit_length = int(sample_rate * bit_duration)
    binary_data = ""

    for i in range(0, len(audio_data), bit_length):
        segment = audio_data[i:i + bit_length]
        # Frequenz analysieren, um festzustellen, ob es sich um ein "0"- oder "1"-Bit handelt
        fft_result = np.fft.fft(segment)
        freqs = np.fft.fftfreq(len(segment), 1 / sample_rate)
        peak_freq = abs(freqs[np.argmax(np.abs(fft_result))])

        if abs(peak_freq - freq_0) < abs(peak_freq - freq_1):
            binary_data += "0"
        else:
            binary_data += "1"

    return binary_data

# Binärdaten in Text umwandeln
def binary_to_text(binary_data):
    text = ""
    for i in range(0, len(binary_data), 8):
        byte = binary_data[i:i + 8]
        if len(byte) == 8:
            text += chr(int(byte, 2))
    return text

# Nachricht entschlüsseln
```

```
def decrypt_message(encrypted_text, key):
    # Nachricht entschlüsseln, indem der Schlüssel mit den ursprünglichen Daten kombiniert wird (XOR)
    decrypted_message = ""
    for i in range(len(encrypted_text)):
        decrypted_char = chr(ord(encrypted_text[i]) ^ ord(key[i % len(key)]))
        decrypted_message += decrypted_char
    return decrypted_message

# Hauptprogramm
if __name__ == "__main__":
    # WAV-Datei lesen
    sample_rate, audio_data = read(user_parameters["wav_filename"])
    if audio_data.ndim > 1:
        audio_data = audio_data[:, 0]  # Falls Stereo, nur einen Kanal verwenden
    audio_data = audio_data / 32767.0  # Normalisieren auf den Bereich [-1, 1]

    # Audiodaten in Binärdaten dekodieren
    binary_data = decode_audio_to_binary(audio_data, user_parameters["bit_duration"], user_parameters["sample_rate"], user_parameters["freq_0"], user_parameters["freq_1"])
    print(f"Binärdaten der Nachricht: {binary_data[:64]}...")  # Zeigt die ersten 64 Bits der Nachricht (für Debugging)

    # Binärdaten in verschlüsselten Text umwandeln
    encrypted_message = binary_to_text(binary_data)

    # Nachricht entschlüsseln
    decrypted_message = decrypt_message(encrypted_message, user_parameters["encryption_key"])
    print(f"Entschlüsselte Nachricht: {decrypted_message}")
```

Auch hier gibt es Parameter, die eingestellt werden können – oder müssen:

```
user_parameters = {
    "sample_rate": 44100,  # Abtastrate (Hz)
    "bit_duration": 0.1,   # Dauer eines Bits (in Sekunden)
    "freq_0": 1000,        # Frequenz für "0" (Hz)
    "freq_1": 2000,        # Frequenz für "1" (Hz)
       "encryption_key": "geheimer_schluessel",   # Verschlüsselungsschlüssel (beide Parteien müssen denselben Schlüssel verwenden)
    "wav_filename": "message.wav"          # Name der zu lesenden WAV-Datei
}
```

Allen voran natürlich der geheime Schlüssel: Dieser muss unbedingt mit jenem übereinstimmen, den der Sender in seinem Script zum Verschlüsseln verwendet hat.

Aber auch die Parameter **sample_rate, bit_duration und die Frequenzen** müssen mit den Einstellungen des Senders übereinstimmen. Der Dateiname der **wav_filename** muss dem Namen der per E-Mail empfangenen Audio-Datei entsprechen – also vor im Script angepasst werden, bevor es ausgeführt wird.

So funktioniert das Script

Das Empfänger-Script liest also die Audiodatei, dekodiert die darin enthaltene Nachricht und entschlüsselt sie. Hier die einzelnen Schritte:

1. **WAV-Datei lesen** Das Script beginnt mit dem Einlesen der Audiodatei mithilfe der Funktion scipy.io.wavfile.read(). Dabei wird die Abtastrate und die Audiodaten extrahiert:

```
                sample_rate, audio_data = read(user_parameters["wav_filename"])
   if audio_data.ndim > 1:
       audio_data = audio_data[:, 0]  # Falls Stereo, nur
```

einen Kanal verwenden
   audio_data = audio_data / 32767.0 # Normalisieren auf den Bereich [-1, 1]

Diese Normalisierung ist notwendig, um die Audiodaten auf einen Bereich zwischen -1 und 1 zu skalieren.

2. **Audiodaten in Binärdaten dekodieren** Die Funktion decode_audio_to_binary() analysiert die Audiodaten und konvertiert sie zurück in eine Binärfolge. Dabei wird die Fourier-Transformation verwendet, um die Frequenzen der einzelnen Segmente zu analysieren und zu entscheiden, ob es sich um ein Bit „0“ oder „1“ handelt:

```
    def decode_audio_to_binary(audio_data, bit_duration, sample_rate, freq_0, freq_1):
        bit_length = int(sample_rate * bit_duration)
        binary_data = ""

        for i in range(0, len(audio_data), bit_length):
            segment = audio_data[i:i + bit_length]
            fft_result = np.fft.fft(segment)
               freqs = np.fft.fftfreq(len(segment), 1 / sample_rate)
                                                peak_freq    = abs(freqs[np.argmax(np.abs(fft_result))])

              if abs(peak_freq - freq_0) < abs(peak_freq - freq_1):
                 binary_data += "0"
            else:
                binary_data += "1"

        return binary_data
```

Diese Funktion durchläuft die Audiodaten in Segmenten und bestimmt für jedes Segment, ob es sich um eine „0“ oder „1“ handelt.

3. **Binärdaten in Text umwandeln** Nachdem die Audiodaten in Binärdaten umgewandelt wurden, werden diese in den ursprünglichen Text konvertiert. Hierbei wird jeder 8-Bit-Block in ein Zeichen umgewandelt:

```
def binary_to_text(binary_data):
    text = ""
    for i in range(0, len(binary_data), 8):
        byte = binary_data[i:i + 8]
        if len(byte) == 8:
            text += chr(int(byte, 2))
    return text
```

So wird der verschlüsselte Text aus den Binärdaten wiederhergestellt.

4. **Nachricht entschlüsseln** Die entschlüsselte Nachricht wird mit der Funktion decrypt_message() wieder in den Klartext umgewandelt. Dazu wird derselbe Schlüssel verwendet, der auch beim Verschlüsseln benutzt wurde:

```
def decrypt_message(encrypted_text, key):
    decrypted_message = ""
    for i in range(len(encrypted_text)):
          decrypted_char = chr(ord(encrypted_text[i]) ^ ord(key[i % len(key)]))
        decrypted_message += decrypted_char
    return decrypted_message
```

Diese Methode führt eine XOR-Operation auf jedes Zeichen des verschlüsselten Textes durch, um die ursprüngliche Nachricht wiederherzustellen.

5. **Ergebnis anzeigen** Schließlich wird die entschlüsselte Nachricht auf der Konsole ausgegeben:

```
print(f"Entschlüsselte Nachricht: {decrypted_message}")
```

Damit erhält der Empfänger die ursprünglich gesendete

Nachricht im Klartext in seinem Terminal:

```
Binärdaten der Nachricht: 00100011000001000001101101000101001110010000101100000000000000000...
Entschlüsselte Nachricht: Das Pferd frisst keinen Gurkensalat.
```

Falls der Empfänger jedoch einen falschen Schlüssel in seinem Script verwendet, klappt es mit dem Entschlüsseln nicht. So sieht die gleiche Nachricht aus, wenn am Ende des Schlüssels zwei Zeichen fehlen:

```
Binärdaten der Nachricht: 00100011000001000001101101000101001110010000101100000000000000000...
Entschlüsselte Nachricht: Das Pferd frisst ilfndf,XOsW~anhjw`8
```

Die Nachricht beginnt zwar korrekt, weil der Anfang des Schlüssels mit jenem des Senders übereinstimmt. Sie „zerfällt“ dann aber zu Kauderwelsch, da der Schlüssel des Empfängers, wie gesagt, zu kurz ist und also zu früh wieder die ersten Zeichen des Schlüssels verwendet werden. **Wenn der Schlüssel überhaupt nicht jenem des Senders entspricht, bleibt die Textnachricht vollständig unlesbar.**

Wie geht es weiter?

Du kannst nun also Nachrichten verschlüsselt als Audio versenden und dir einigermaßen sicher sein, dass sie nur jemand entschlüsseln kann, der ein geeignetes Script und vor allem den richtigen Schlüssel dafür besitzt. Wie könnten Verbesserungen aussehen? Du hast vielleicht schon bemerkt, dass die erzeugten WAV-Dateien recht groß sind – die relative kurze Nachricht aus dem Beispiel oben hat bereits 2,6 MB.

Hier könnte eine Konvertierung in das MP3-Format weiterhelfen, um sicherzugehen, dass deine Nachricht nicht zu groß für den Anhang einer E-Mail ist.