Audio Steganography

Phase 02 A Week 01

INTRODUCTION:

In today's digital landscape, the threat of covert communication and hidden malicious activities poses a significant challenge to cybersecurity. Attackers employ various techniques to bypass security measures and infiltrate systems, one of which is steganography. Steganography involves concealing secret information within seemingly harmless files, such as images, audio, or videos, making it incredibly difficult to detect without specialized tools. The ability to hide malicious contents within the vast amount of digital media exchanged daily presents a significant concern for cybersecurity professionals.

The importance of steganography lies in its ability to provide a covert means of transmitting malicious code or commands to victim machines. By embedding the malicious content within innocent-looking files, threat actors can exploit the trust placed in these files and evade detection by traditional security measures, including antivirus software. This technique enables them to establish a foothold in the victim's environment, execute malicious operations, and potentially exfiltrate sensitive data.

LSB (Least Significant Bit) implementation technique:

LSB implementation in steganography is a widely used technique for embedding secret information within digital media files. It takes advantage of the fact that altering the least significant bits of pixel values in an image or audio sample generally does not result in a noticeable change to human perception.

In this attack vector, the attacker manipulates the LSBs of selected pixels or samples in the cover media (such as an image or audio file) to encode the secret information. The LSBs are modified to represent the bits of the secret message, effectively hiding the information within the cover media.

However, there are limitations and challenges associated with LSB steganography. One of the main limitations is the trade-off between payload capacity and the detectability of the hidden information. As more LSBs are altered, the potential for detectable distortion increases. Additionally, advanced steganalysis techniques and algorithms have been developed to detect LSB manipulations, posing a risk to the effectiveness of this technique.

Despite its limitations, LSB implementation in steganography remains a prevalent attack vector due to its simplicity and the difficulty in detecting hidden information.

To work on this attack vector, we wrote 2 simple C codes which performs LSB implementation on audio files and analysed its behaviour. Compared to images, audio files provide the advantage of accommodating a greater amount of malicious content. This is primarily due to their larger file size, allowing for more data to be hidden within the audio data.

Audio Steganography

Phase 02 A Week 01

Implementation:

1. **Embedder sample**:

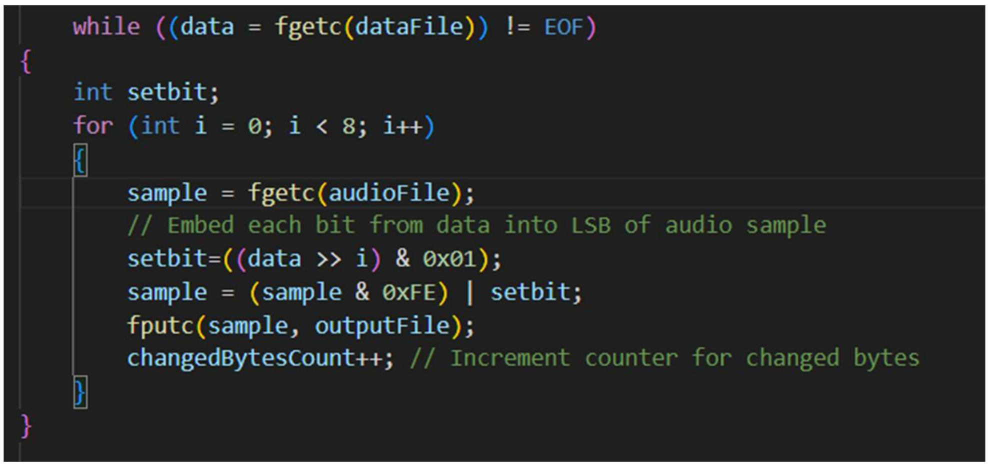
   This C program employs a simple LSB technique to embed a text file contents into an audio file. This code is executed on the attacker side and the altered audio file could be hosted on the attacker's servers or infected servers.

   Initially it creates an output audio file and copies the header contents into the output audio file. This is done to ensure that the output audio is recognised as audio file by the OS (since header contains signature and important information about files). Next, if follows the following algorithm to embed the contents of 'data.txt' into 'output.m4a' file.

   - The algorithm begins by reading a byte of data from the 'data.txt' file using the fgetc() function. The data is stored in the 'data' variable.
   - The algorithm then enters a loop that iterates 8 times, representing each bit of the 'data' byte.
   - Inside the loop, it reads a byte from the audio file using the fgetc() function. This byte represents a sample of the audio data.
   - The next step is to embed each bit from the 'data' byte into the Least Significant Bit (LSB) of the audio sample. This is done by performing a bitwise right shift on the 'data' byte by the current bit position 'i', and then using a bitwise AND operation with 0x01 to extract the LSB. The result is stored in the 'setbit' variable.
   - To embed the bit into the audio sample, the algorithm first clears the LSB of the sample by performing a bitwise AND operation with 0xFE, which sets the LSB to 0. Then, it performs a bitwise OR operation with 'setbit', which sets the LSB to the value of the bit from the 'data' byte.
   - The modified audio sample is then written to the output file using the fputc() function.
   - The loop continues for each bit of the data byte until all bits have been embedded into the audio samples.

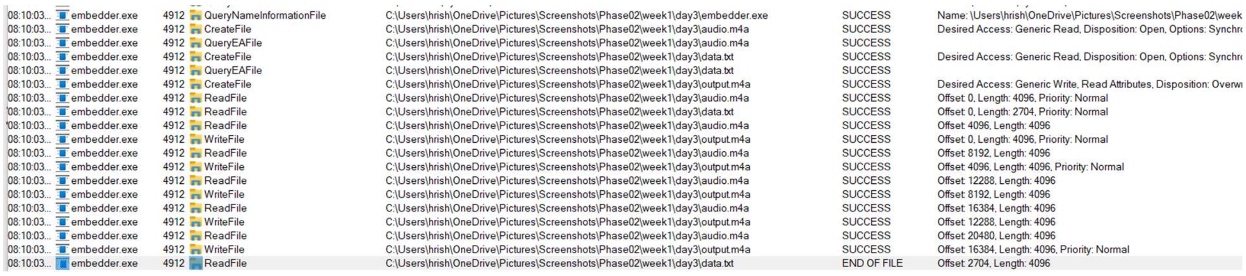
- The algorithm repeats this process until the end of the data file is reached.

The C snippet for the algorithm is given below:



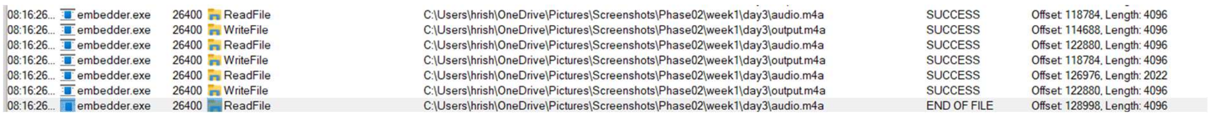
```c
    while ((data = fgetc(dataFile)) != EOF)
{
    int setbit;
    for (int i = 0; i < 8; i++)
    {
        sample = fgetc(audioFile);
        // Embed each bit from data into LSB of audio sample
        setbit=((data >> i) & 0x01);
        sample = (sample & 0xFE) | setbit;
        fputc(sample, outputFile);
        changedBytesCount++; // Increment counter for changed bytes
    }
}
```

When we capture the events in Procmon, we see the following:



```
08:10:03...  embedder.exe  4912  QueryNameInformationFile  C:\Users\hrish\OneDrive\Pictures\Screenshots\Phase02\week1\day3\embedder.exe  SUCCESS      Name: \Users\hrish\OneDrive\Pictures\Screenshots\Phase02\week
08:10:03...  embedder.exe  4912  CreateFile                C:\Users\hrish\OneDrive\Pictures\Screenshots\Phase02\week1\day3\audio.m4a     SUCCESS      Desired Access: Generic Read, Disposition: Open, Options: Synchr
08:10:03...  embedder.exe  4912  QueryEAFile               C:\Users\hrish\OneDrive\Pictures\Screenshots\Phase02\week1\day3\audio.m4a     SUCCESS
08:10:03...  embedder.exe  4912  CreateFile                C:\Users\hrish\OneDrive\Pictures\Screenshots\Phase02\week1\day3\data.txt      SUCCESS      Desired Access: Generic Read, Disposition: Open, Options: Synchr
08:10:03...  embedder.exe  4912  QueryEAFile               C:\Users\hrish\OneDrive\Pictures\Screenshots\Phase02\week1\day3\data.txt      SUCCESS
08:10:03...  embedder.exe  4912  CreateFile                C:\Users\hrish\OneDrive\Pictures\Screenshots\Phase02\week1\day3\output.m4a    SUCCESS      Desired Access: Generic Write, Read Attributes, Disposition: Overw
08:10:03...  embedder.exe  4912  ReadFile                  C:\Users\hrish\OneDrive\Pictures\Screenshots\Phase02\week1\day3\audio.m4a     SUCCESS      Offset: 0, Length: 4096, Priority: Normal
08:10:03...  embedder.exe  4912  ReadFile                  C:\Users\hrish\OneDrive\Pictures\Screenshots\Phase02\week1\day3\data.txt      SUCCESS      Offset: 0, Length: 2704, Priority: Normal
08:10:03...  embedder.exe  4912  ReadFile                  C:\Users\hrish\OneDrive\Pictures\Screenshots\Phase02\week1\day3\audio.m4a     SUCCESS      Offset: 4096, Length: 4096
08:10:03...  embedder.exe  4912  WriteFile                 C:\Users\hrish\OneDrive\Pictures\Screenshots\Phase02\week1\day3\output.m4a    SUCCESS      Offset: 0, Length: 4096, Priority: Normal
08:10:03...  embedder.exe  4912  ReadFile                  C:\Users\hrish\OneDrive\Pictures\Screenshots\Phase02\week1\day3\audio.m4a     SUCCESS      Offset: 8192, Length: 4096
08:10:03...  embedder.exe  4912  WriteFile                 C:\Users\hrish\OneDrive\Pictures\Screenshots\Phase02\week1\day3\output.m4a    SUCCESS      Offset: 4096, Length: 4096, Priority: Normal
08:10:03...  embedder.exe  4912  ReadFile                  C:\Users\hrish\OneDrive\Pictures\Screenshots\Phase02\week1\day3\audio.m4a     SUCCESS      Offset: 12288, Length: 4096
08:10:03...  embedder.exe  4912  WriteFile                 C:\Users\hrish\OneDrive\Pictures\Screenshots\Phase02\week1\day3\output.m4a    SUCCESS      Offset: 8192, Length: 4096
08:10:03...  embedder.exe  4912  ReadFile                  C:\Users\hrish\OneDrive\Pictures\Screenshots\Phase02\week1\day3\audio.m4a     SUCCESS      Offset: 16384, Length: 4096
08:10:03...  embedder.exe  4912  WriteFile                 C:\Users\hrish\OneDrive\Pictures\Screenshots\Phase02\week1\day3\output.m4a    SUCCESS      Offset: 12288, Length: 4096
08:10:03...  embedder.exe  4912  ReadFile                  C:\Users\hrish\OneDrive\Pictures\Screenshots\Phase02\week1\day3\audio.m4a     SUCCESS      Offset: 20480, Length: 4096
08:10:03...  embedder.exe  4912  WriteFile                 C:\Users\hrish\OneDrive\Pictures\Screenshots\Phase02\week1\day3\output.m4a    SUCCESS      Offset: 16384, Length: 4096, Priority: Normal
08:10:03...  embedder.exe  4912  ReadFile                  C:\Users\hrish\OneDrive\Pictures\Screenshots\Phase02\week1\day3\data.txt      END OF FILE  Offset: 2704, Length: 4096
```

We see read and write events between ‘audio.m4a’ and ‘output.m4a’ files until the EOF of the ‘data.txt’ is reached. Now, the embedder program just copies the remain audio file contents of ‘audio.m4a’ until the end of file for it is reached. The same can be seen in the screenshot below:



```
08:16:26...  embedder.exe  26400  ReadFile   C:\Users\hrish\OneDrive\Pictures\Screenshots\Phase02\week1\day3\audio.m4a   SUCCESS      Offset: 118784, Length: 4096
08:16:26...  embedder.exe  26400  WriteFile  C:\Users\hrish\OneDrive\Pictures\Screenshots\Phase02\week1\day3\output.m4a  SUCCESS      Offset: 114688, Length: 4096
08:16:26...  embedder.exe  26400  ReadFile   C:\Users\hrish\OneDrive\Pictures\Screenshots\Phase02\week1\day3\audio.m4a   SUCCESS      Offset: 122880, Length: 4096
08:16:26...  embedder.exe  26400  WriteFile  C:\Users\hrish\OneDrive\Pictures\Screenshots\Phase02\week1\day3\output.m4a  SUCCESS      Offset: 118784, Length: 4096
08:16:26...  embedder.exe  26400  ReadFile   C:\Users\hrish\OneDrive\Pictures\Screenshots\Phase02\week1\day3\audio.m4a   SUCCESS      Offset: 126976, Length: 2022
08:16:26...  embedder.exe  26400  WriteFile  C:\Users\hrish\OneDrive\Pictures\Screenshots\Phase02\week1\day3\output.m4a  SUCCESS      Offset: 122880, Length: 4096
08:16:26...  embedder.exe  26400  ReadFile   C:\Users\hrish\OneDrive\Pictures\Screenshots\Phase02\week1\day3\audio.m4a   END OF FILE  Offset: 128998, Length: 4096
```

2. **Extractor sample**:

   This C program uses the same mechanism in reverse to extract the contents from the audio file and write the extracted contents into a text file. This program will be placed in victim’s machine. So, when victim let us say downloads the altered audio file from attacker’s server, with extractor program or malware in the victim’s machine, attacker can extract the malicious contents from the audio file and execute them. However, in this extractor program, we simply write the extracted contents into a text file.

Initially, the program skips the header part of the altered audio file, opens a text file to write the extracted contents and starts the following algorithm.

- The code operates within a ‘while’ loop that continues until the end of the audio file is reached. Each byte of the audio file is read using ‘fgetc(audioFile)’ and stored in the ‘sample’ variable.

- The least significant bit (LSB) of the sample is extracted by performing a bitwise AND operation (sample & 0x01) with 0x01, which isolates the least significant bit.

- The extracted bit is appended to the ‘data’ variable using a bitwise left shift (data << 1) and bitwise OR (|) operation. This accumulates the previously extracted bits with the new bit.

- The bitCount is incremented to keep track of the number of bits extracted. If bitCount reaches 8 (indicating that 8 bits have been accumulated in data), the code proceeds with the following steps:

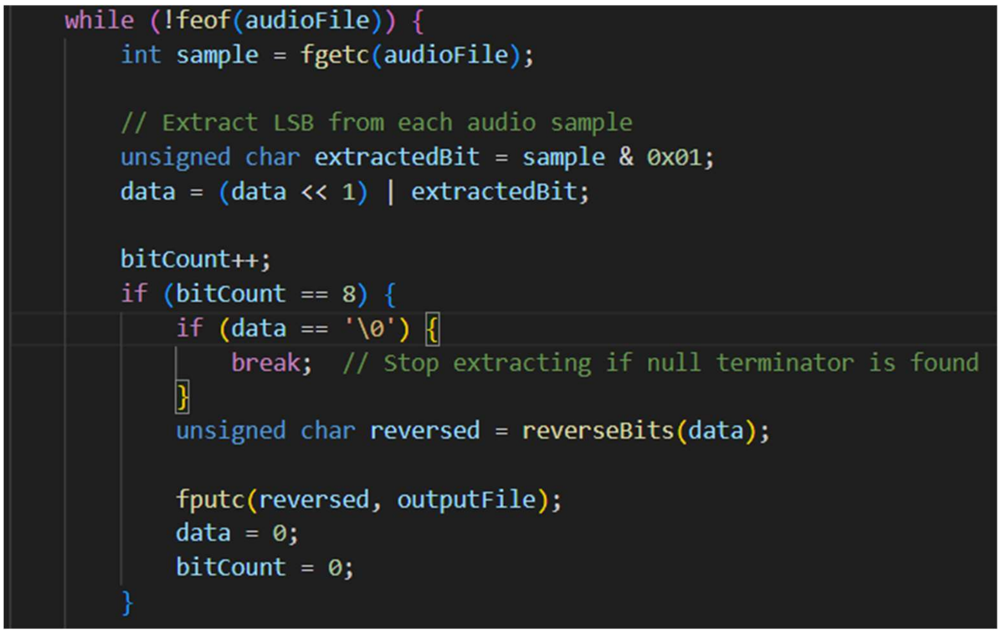
    - It checks if the accumulated data is equal to a null terminator ('\0'). If it is, the extraction process is stopped by breaking out of the loop.

    - If the null terminator is not found, the data is passed through a function called ’reverseBits’, which reverses the order of the bits. The result is stored in the ‘reversed’ variable. The reason for reversing is that, while embedding, the contents of the text file are read from left to write , top to bottom. But the contents in a file are stored in Little Endian Order (so ‘F’ letter is stored as ‘01100010’ and not as ‘01000110’). But while embedding, we embed the contents in Big Endian Order. Hence, while extracting also we extract them in Big Endian Order. So we must reverse the contents while storing in a text file.

    - The reversed value is written to the outputFile using fputc. The data variable and the bitCount are reset to 0 to prepare for the extraction of the next 8 bits.

- The loop continues until the null terminator is found or the end of the audio file is reached.

Following is the code for extraction:



```
while (!feof(audioFile)) {
    int sample = fgetc(audioFile);

    // Extract LSB from each audio sample
    unsigned char extractedBit = sample & 0x01;
    data = (data << 1) | extractedBit;

    bitCount++;
    if (bitCount == 8) {
        if (data == '\0') {
            break;  // Stop extracting if null terminator is found
        }
        unsigned char reversed = reverseBits(data);

        fputc(reversed, outputFile);
        data = 0;
        bitCount = 0;
    }
```

In conclusion, after verifying the contents of the extracted text file, we have successfully completed the implementation of the LSB (Least Significant Bit) extraction technique. However, it's worth noting that the code sample represents a simple implementation of LSB extraction and may not incorporate some of the advanced techniques used in modern steganography and data hiding practices. Here are a few advanced techniques that are commonly employed:

- **Spread Spectrum Techniques**: These techniques involve spreading the hidden data across multiple bits or samples rather than just a single LSB.
- **Adaptive Data Hiding**: This approach dynamically adjusts the embedding strength based on the characteristics of the cover media. It takes into account the perceptual model of the human sensory system to determine optimal locations for embedding data while minimizing the perceptual impact on the cover media.
- **Error Diffusion Techniques**: These techniques distribute the embedding error caused by modifying the cover media across neighboring samples or pixels. This helps to minimize visual artifacts and statistical detectability in the stego media.

These advanced techniques aim to improve the security, robustness, and perceptual quality of the hidden data. Implementing such techniques requires a deeper understanding of signal processing, coding theory, and perceptual models. They are commonly utilized in modern steganographic applications to achieve a balance between concealment and detection resistance.