Chapter 2: Hidden Threads - Mastering the Art of Steganography in Rust

Introduction

At the core of this fascinating chapter lies the perplexing art of **steganography**, a term derived from the fusion of two ancient Greek words - “steganos”, meaning “to cover or protect,” and “graphien”, signifying “to write.” In the vast domain of **cybersecurity**, steganography serves as a captivating technique, like a digital cover, hiding valuable data within the seemingly innocent façade of images. This practice has become routine within the security community, a skill where practitioners adeptly navigate the delicate balance between hiding and disclosure. In essence, steganography empowers security professionals to embed payloads secretly, patiently awaiting the opportune moment for extraction once the data reaches its intended destination.

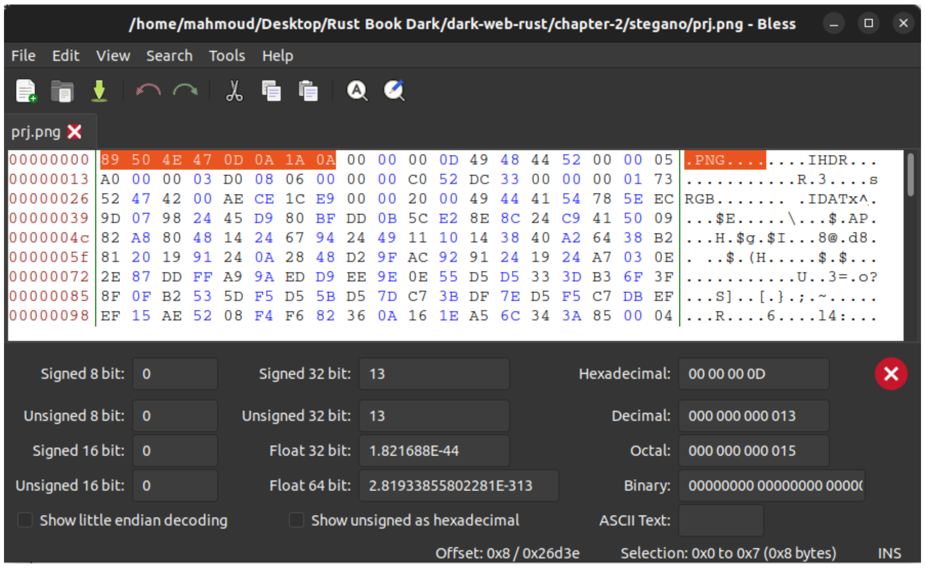
As we kick off this chapter through the complexities of steganography, our focus sharpens on the security landscape, where the art of hiding and revealing data takes center stage. The essence of this practice lies in the ingenious hiding of information within the very fabric of images. Like a hidden message within a painting, steganography allows security practitioners to obscure critical payloads within the pixels of images, laying the groundwork for secret communication. This chapter, which is like a roadmap through the hidden passages of digital secrecy, reveals the techniques and procedures employed in this confidential art, offering insights into the methods that security professionals employ to navigate this delicate dance of data manipulation.

The chapter, like a guidebook for digital spies, delves into the practical aspects of this obscuring art, particularly focusing on the embedding of data within **Portable Network Graphics (PNG)** images. The PNG format, known for its ubiquity and versatility, becomes the canvas upon which the steganographer paints hidden messages. The journey through this chapter promises a deep dive into the complexities of PNG files, exploring their byte structure, decoding headers, and cracking the sequence of chunks that define the anatomy of these digital canvases. In essence, the chapter stands as a beacon for those navigating the maze of steganography, offering both theoretical insights and practical wisdom for concealing and unveiling secrets within the digital realm.

1. Exploring the PNG Format

To kick off our journey of embedding data within the PNG format, we must first understand the complex byte curtain that constitutes a binary PNG image file. The PNG specification, accessible at **libpng.org**, becomes our compass in navigating this digital landscape. Within this knowledge, byte chunks incorporate the fabric of PNG images, creating a canvas where our steganographic secrets will find their hiding places. These chunks, repetitive in nature, lay the groundwork for our exploration into the hidden domains of data manipulation.

The Enigmatic Header:


/home/mahmoud/Desktop/Rust Book Dark/dark-web-rust/chapter-2/stegano/prj.png - Bless

```
00000000 89 50 4E 47 0D 0A 1A 0A 00 00 00 0D 49 48 44 52 00 00 05 .PNG........IHDR...
00000013 A0 00 00 03 D0 08 06 00 00 00 C0 52 DC 33 00 00 00 01 73 ...........R.3....s
00000026 52 47 42 00 AE CE 1C E9 00 00 20 00 49 44 41 54 78 5E EC RGB........ .IDATx^.
00000039 9D 07 98 24 45 D9 80 BF DD 0B 5C E2 8E 8C 24 C9 41 50 09 ...$E.....\...$.AP.
0000004c 82 A8 80 48 14 24 67 94 24 49 11 10 14 38 40 A2 64 38 B2 ...H.$g.$I...8@.d8.
0000005f 81 20 19 91 24 0A 28 48 D2 9F AC 92 91 24 19 24 A7 03 0E .  ..$.(H.....$.$...
00000072 2E 87 DD FF A9 9A ED D9 EE 9E 0E 55 D5 D5 33 3D B3 6F 3F ...........U..3=.o?
00000085 8F 0F B2 53 5D F5 D5 5B D5 7D C7 3B DF 7E D5 F5 C7 DB EF ...S]..[.}.;.~.....
00000098 EF 15 AE 52 08 F4 F6 82 36 0A 16 1E A5 6C 34 3A 85 00 04 ...R....6....l4:...
```

```
Signed 8 bit: 0          Signed 32 bit: 13                   Hexadecimal: 00 00 00 0D
Unsigned 8 bit: 0        Unsigned 32 bit: 13                 Decimal: 000 000 000 013
Signed 16 bit: 0         Float 32 bit: 1.821688E-44          Octal: 000 000 000 015
Unsigned 16 bit: 0       Float 64 bit: 2.81933855802281E-313 Binary: 00000000 00000000 0000(
Show little endian decoding   Show unsigned as hexadecimal   ASCII Text:
Offset: 0x8 / 0x26d3e   Selection: 0x0 to 0x7 (0x8 bytes)   INS
```

Figure 1: ScreenShot taken from the Bless hex editor

In the realm of PNG files, our journey begins with the opening eight bytes, forming the magical sequence "**89 50 4E 47 0D 0A 1A 0A**". This sequence acts as a secret code, the key that unlocks the PNG world. Picture it like the first musical notes of our steganographic symphony. Now, onto the scene comes the `Header` struct, a kind of container designed to catch these magical bytes.

```rust
// Header holds the first UINT64 (Magic Bytes)
struct Header {
    header: u64,
}
```

The `Header` structure is like an artist's canvas waiting for a masterpiece. It holds an unsigned 64-bit integer, a perfect fit for the magical bytes. When we perform a magical act called **transmutation**, the hidden message "PNG" is revealed, almost like a secret handshake granting us access to the **PNG** world. This Header is our guardian, making sure our PNG image is the real deal and giving us the green light to start our work of steganography. It's like the keeper of secrets, ensuring the symphony begins without any interference.

Decoding the Chunk Sequence:

As we venture beyond the surface of the PNG image, we encounter something known as the chunk sequence. This sequence follows structured data of **SIZE** (4 bytes), **TYPE** (4 bytes), **DATA** (variable bytes), and **CRC** (4 bytes). Each chunk is like a puzzle piece in the steganographic mosaic and contributes its unique narrative to the overall story painted within the **PNG** image. The **SIZE** chunk sets the stage, dictating the length of the upcoming **DATA**, while the **TYPE** chunk acts as a sign, revealing the purpose of the data that follows. Our mission is to decode this sequence, uncovering the hidden messages that lie beneath.

```
// Chunk represents a data byte chunk segment
struct Chunk {
    size: u32,
    r#type: u32,
    data: Vec<u8>,
    crc: u32,
}
```

As we embark on the complexities of steganography within the PNG image, the `MetaChunk` structure emerges as our reliable companion, illuminating the path through the concealed landscape of digital storytelling. Picture the `MetaChunk` as our guide, equipped with the tools to decode the complexities hidden within each segment of our steganographic work.

```
// MetaChunk structure orchestrating the steganographic journey
struct MetaChunk {
    header: Header,
    chk: Chunk,
    offset: u64,
}
```

The `MetaChunk` will navigate us through the hidden landscapes of binary complexities. Its fields, such as the `header` and `chk` (representing the Header and Chunk structures, respectively), are our compass and map, ensuring we stay on course through this steganographic work. The `offset`, is like a GPS coordinate, pinpoints our location within the byte landscape, marking the starting point of each chunk segment.

2. Reading a PNG File

Preprocessing the Image: The `pre_process_image` function, our initiation into the PNG world, transforms the PNG image into a readable script. It's the first step in turning a PNG image, which seems like a puzzle, into something we can read and work with. This function does some important things to make that happen.

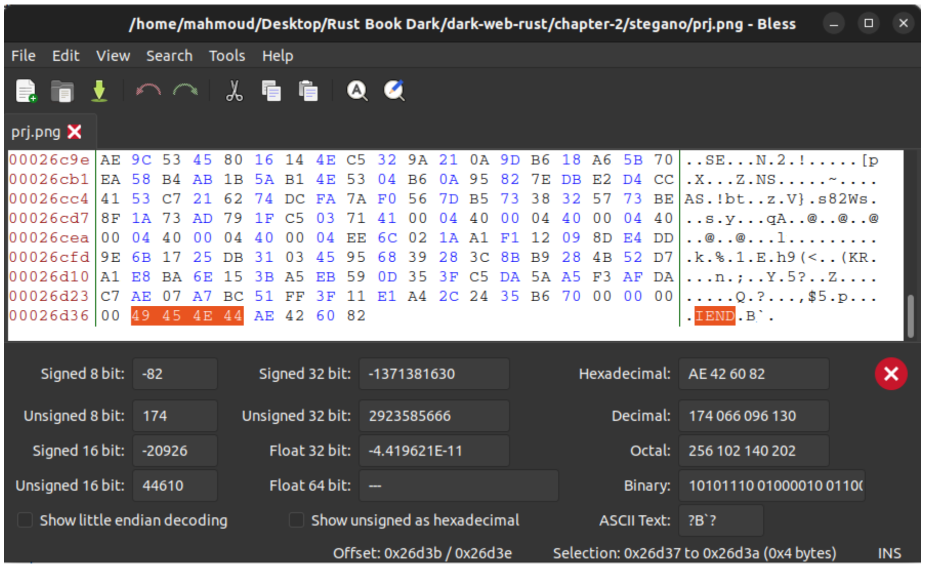
```
// Implementation of MetaChunk, defining a function `pre_process_image` that processes a PN
impl MetaChunk {
    fn pre_process_image(file: &mut File) -> Result<MetaChunk, std::io::Error> {
        // Creating a Header struct to hold the first 8 bytes of the PNG image
        let mut header = Header { header: 0 };
        // Reading exactly 8 bytes from the file into the header using unsafe operations
        file.read_exact(unsafe { mem::transmute::<_, &mut [u8; 8]>(&mut header.header) })?;

        // Converting the header bytes from u64 to u8 array
        let b_arr = u64_to_u8_array(header.header);

        // Checking if the PNG magic bytes are present in the file
        if &b_arr[1..4] != b"PNG" {
            // If not, panicking and terminating the program with an error message
            panic!("Provided file is not a valid PNG format");
        } else {
            // If yes, printing a confirmation message
            println!("It is a valid PNG file. Let's process it!");
        }

        // Getting the current position (offset) in the file and storing it
        let offset = file.seek(SeekFrom::Current(0))?;

        // Returning a `MetaChunk` struct instance with the obtained header, an initial Chu
        Ok(MetaChunk {
            header,
            chk: Chunk {
                size: 0,
                r#type: 0,
                data: Vec::new(),
                crc: 0,
            },
            offset,
        })
    }
}
```

Once initiated, validation becomes crucial. This method inspect the first eight bytes, ensuring their adherence to the PNG format. The magic bytes unfold, and if the decoded message aligns with "PNG", the gateway to our steganographic exploration swings open.

Reading through Chunk Sequences:

Our journey into the PNG landscape is marked by the `process_image` function, a blend of code orchestrating the navigation through chunks. A loop, a recurring motif, guides the traversal through each chunk. The `count` becomes our companion, marking each chunk's unveiling in this steganographic manuscript.

```
fn process_image(&mut self, file: &mut File) {
    let mut count = 1;
    let mut chunk_type = String::new();
    let end_chunk_type = "IEND";
```


/home/mahmoud/Desktop/Rust Book Dark/dark-web-rust/chapter-2/stegano/prj.png - Bless
File Edit View Search Tools Help
prj.png

```
00026c9e AE 9C 53 45 80 16 14 4E C5 32 9A 21 0A 9D B6 18 A6 5B 70 ..SE...N.2.!.....[p
00026cb1 EA 58 B4 AB 1B 5A B1 4E 53 04 B6 0A 95 82 7E DB E2 D4 CC .X...Z.NS.....~....
00026cc4 41 53 C7 21 62 74 DC FA 7A F0 56 7D B5 73 38 32 57 73 BE AS.!bt..z.V}.s82Ws.
00026cd7 8F 1A 73 AD 79 1F C5 03 71 41 00 04 40 00 04 40 00 04 40 ..s.y...qA..@..@..@
00026cea 00 04 40 00 04 40 00 04 EE 6C 02 1A A1 F1 12 09 8D E4 DD ..@..@...l.........
00026cfd 9E 6B 17 25 DB 31 03 45 95 68 39 28 3C 8B B9 28 4B 52 D7 .k.%.1.E.h9(<..(KR.
00026d10 A1 E8 BA 6E 15 3B A5 EB 59 0D 35 3F C5 DA 5A A5 F3 AF DA ...n.;..Y.5?..Z....
00026d23 C7 AE 07 A7 BC 51 FF 3F 11 E1 A4 2C 24 35 B6 70 00 00 00 .....Q.?...,$5.p...
00026d36 00 49 45 4E 44 AE 42 60 82                               .IEND.B`.
```

Signed 8 bit: -82
Signed 32 bit: -1371381630
Hexadecimal: AE 42 60 82
Unsigned 8 bit: 174
Unsigned 32 bit: 2923585666
Decimal: 174 066 096 130
Signed 16 bit: -20926
Float 32 bit: -4.419621E-11
Octal: 256 102 140 202
Unsigned 16 bit: 44610
Float 64 bit: ---
Binary: 10101110 01000010 0110(
Show little endian decoding
Show unsigned as hexadecimal
ASCII Text: ?B`?
Offset: 0x26d3b / 0x26d3e
Selection: 0x26d37 to 0x26d3a (0x4 bytes)
INS

Figure 2: ScreenShot taken from the Bless hex editor

```rust
    while chunk_type != end_chunk_type {
        println!("---- Chunk # {} ----", count);
        let offset = self.get_offset(file);
        println!("Chunk Offset: {:x}", offset);
        self.read_chunk(file);
        chunk_type = self.chunk_type_to_string();
        count += 1;
    }
}
```

Charting Coordinates with Offset:

In this exploration, awareness of our position within the PNG image becomes imperative. The `get_offset` method, a virtual compass, captures the coordinates of each chunk. The **`seek`** function, a navigator's tool, unfolds the map of offsets, guiding our steganographic ship through the PNG terrain. It moves to an offset, in bytes, in the file stream.

```rust
fn get_offset(&mut self, data: &mut File) -> u64 {
    let offset = data.seek(SeekFrom::Current(0)).unwrap();
    self.offset = offset;
    offset
}
```

Deciphering Chunk by Chunk:

The heart of our steganographic novel lies in the `read_chunk` method—a meticulous deciphering of each chunk's data. The synchronization of `read_chunk_size`, `read_chunk_type`, `read_chunk_bytes`, and `read_chunk_crc` becomes a choreography of bytes, each revealing a distinct aspect of the hidden bytes.

```rust
fn read_chunk(&mut self, data: &mut File) {
    self.read_chunk_size(data);
    self.read_chunk_type(data);
    self.read_chunk_bytes(data, self.chk.size);
    self.read_chunk_crc(data);
}
```

Transcribing the Size Chunk:

The `read_chunk_size` method reads the size of each chunk. The `size_bytes` variable holds the essence of the chunk size on the paper of our steganographic manuscript.

```rust
fn read_chunk_size(&mut self, data: &mut File) {
    let mut size_bytes = [0; 4];

    match data.read_exact(&mut size_bytes) {
        Ok(_) => {
            self.chk.size = u32::from_be_bytes(size_bytes);
        }
        Err(err) if err.kind() == ErrorKind::UnexpectedEof => {
            eprintln!("Warning: Reached end of file prematurely while reading chunk size");
        }
        Err(err) => {
            eprintln!("Error reading chunk size bytes: {}", err);
        }
    }
}
```

Decoding the Chunk Type: The `read_chunk_type` method deciphers the type of each chunk.

```rust
fn read_chunk_type(&mut self, data: &mut File) {
    let mut type_bytes = [0; 4];

    match data.read_exact(&mut type_bytes) {
        Ok(_) => {
            self.chk.r#type = u32::from_be_bytes(type_bytes);
        }
        Err(err) if err.kind() == ErrorKind::UnexpectedEof => {
            eprintln!("Warning: Reached end of file prematurely while reading chunk type");
        }
        Err(err) => {
            eprintln!("Error reading chunk type bytes: {}", err);
        }
    }
}
```

Reading the Chunk's Bytes: The `read_chunk_bytes` method reads each chunk of the image file given a number of bytes.

```
fn read_chunk_bytes(&mut self, data: &mut File, len: u32) {
    self.chk.data = vec![0; len as usize];

    match data.read_exact(&mut self.chk.data) {
        Ok(_) => {
            // Successfully read the expected number of bytes
        }
        Err(err) if err.kind() == ErrorKind::UnexpectedEof => {
            eprintln!("Error reading chunk bytes: Reached end of file prematurely");
            self.chk
                .data
                .truncate(data.seek(SeekFrom::Current(0)).unwrap() as usize);
        }
        Err(err) => {
            eprintln!("Error reading chunk bytes: {}", err);
        }
    }
}
```

Decoding the Chunk's CRC: The `read_chunk_crc` method takes on the role of the guardian, deciphering the script's integrity. The CRC bytes ensure that the essence of the chunk remains unaltered.

```
fn read_chunk_crc(&mut self, data: &mut File) {
    let mut crc_bytes = [0; 4];

    match data.read_exact(&mut crc_bytes) {
        Ok(_) => {
            self.chk.crc = u32::from_be_bytes(crc_bytes);
        }
        Err(err) if err.kind() == ErrorKind::UnexpectedEof => {
            eprintln!("Warning: Reached end of file prematurely while reading chunk CRC");
        }
        Err(err) => {
            eprintln!("Error reading chunk CRC bytes: {}", err);
        }
    }
}
```

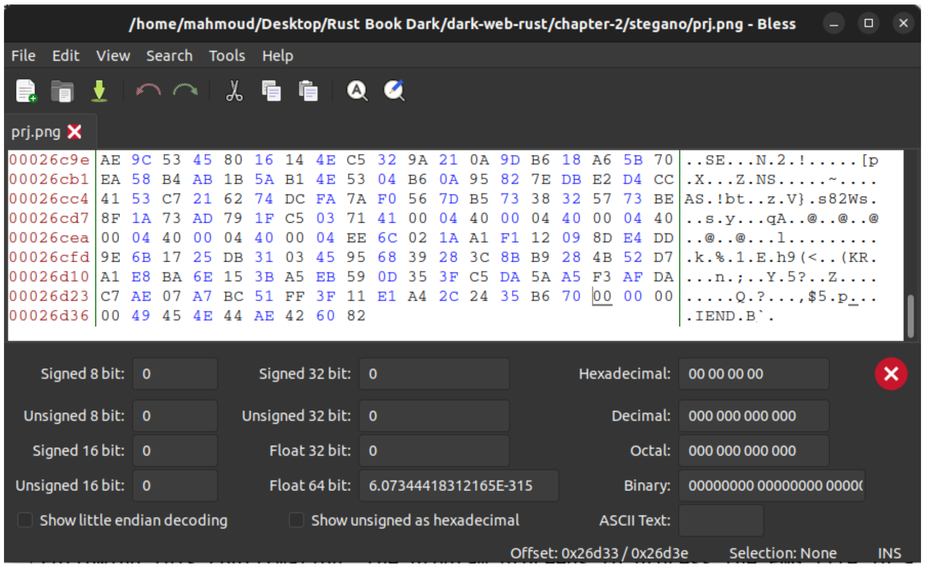
As the stenographic odyssey unfolds, each line of code becomes a step in deciphering the hidden canvas of PNG images. The orchestration of structs, functions, and bytes creates a symphony that resonates with the heartbeat of steganography. With every offset, chunk, and byte, the cover over PNG's secrets slowly lifts, revealing a digital narrative waiting to be explored.

Now, let's put it all together.

```rust
use std::fs::File;
use std::io::ErrorKind;
use std::io::{Read, Seek, SeekFrom};
use std::mem;
use std::str;

#[derive(Debug)]
struct Header {
    header: u64,
}

#[derive(Debug)]
struct Chunk {
    size: u32,
    r#type: u32,
    data: Vec<u8>,
    crc: u32,
}

struct MetaChunk {
    header: Header,
    chk: Chunk,
    offset: u64,
}

fn u64_to_u8_array(value: u64) -> [u8; 8] {
    let bytes = value.to_ne_bytes();
    let mut result = [0; 8];

    unsafe {
        result = mem::transmute_copy(&bytes);
    }

    result
}

impl MetaChunk {
    fn pre_process_image(file: &mut File) -> Result<MetaChunk, std::io::Error> {
        let mut header = Header { header: 0 };
        file.read_exact(unsafe { mem::transmute::<_, &mut [u8; 8]>(&mut header.header) })?;

        let b_arr = u64_to_u8_array(header.header);
        if &b_arr[1..4] != b"PNG" {
            panic!("Not a valid PNG format");
        } else {
            println!("It is a valid PNG file. Let's process it!");
        }

        let offset = file.seek(SeekFrom::Current(0))?;
        Ok(MetaChunk {
            header,
            chk: Chunk {
                size: 0,
                r#type: 0,
                data: Vec::new(),
                crc: 0,
            },
            offset,
        })
    }

    fn process_image(&mut self, file: &mut File) {
        let mut count = 1;
        let mut chunk_type = String::new();
        let end_chunk_type = "IEND";

        while chunk_type != end_chunk_type {
            println!("---- Chunk # {} ----", count);
            let offset = self.get_offset(file);
            println!("Chunk Offset: {:x}", offset);
            self.read_chunk(file);
            chunk_type = self.chunk_type_to_string();
            count += 1;
        }
    }

    fn get_offset(&mut self, file: &mut File) -> u64 {
        let offset = file.seek(SeekFrom::Current(0)).unwrap();
        self.offset = offset;
        offset
    }

    fn read_chunk(&mut self, file: &mut File) {
        self.read_chunk_size(file);
        self.read_chunk_type(file);
        self.read_chunk_bytes(file, self.chk.size);
        self.read_chunk_crc(file);
    }

    fn read_chunk_size(&mut self, file: &mut File) {
        let mut size_bytes = [0; 4];

        match file.read_exact(&mut size_bytes) {
            Ok(_) => {
                // Successfully read the expected number of bytes
                self.chk.size = u32::from_be_bytes(size_bytes);
            }
            Err(err) if err.kind() == ErrorKind::UnexpectedEof => {
                // Handle the situation where the file ends before reading the expected byt
                eprintln!("Warning: Reached end of file prematurely while reading chunk siz
            }
            Err(err) => {
                eprintln!("Error reading chunk size bytes: {}", err);
            }
        }
    }

    fn read_chunk_type(&mut self, file: &mut File) {
        let mut type_bytes = [0; 4];

        match file.read_exact(&mut type_bytes) {
            Ok(_) => {
                // Successfully read the expected number of bytes
                self.chk.r#type = u32::from_be_bytes(type_bytes);
            }
            Err(err) if err.kind() == ErrorKind::UnexpectedEof => {
                // Handle the situation where the file ends before reading the expected byt
                eprintln!("Warning: Reached end of file prematurely while reading chunk typ
            }
            Err(err) => {
                eprintln!("Error reading chunk type bytes: {}", err);
            }
        }
    }

    fn read_chunk_bytes(&mut self, file: &mut File, len: u32) {
        self.chk.data = vec![0; len as usize];

        match file.read_exact(&mut self.chk.data) {
            Ok(_) => {
                // Successfully read the expected number of bytes
            }
            Err(err) if err.kind() == ErrorKind::UnexpectedEof => {
                eprintln!("Error reading chunk bytes: Reached end of file prematurely");
                // Update the length of the Chunk based on the actual number of bytes read
                self.chk
                    .data
                    .truncate(file.seek(SeekFrom::Current(0)).unwrap() as usize);
            }
            Err(err) => {
                eprintln!("Error reading chunk bytes: {}", err);
            }
        }
    }

    fn read_chunk_crc(&mut self, file: &mut File) {
        let mut crc_bytes = [0; 4];

        match file.read_exact(&mut crc_bytes) {
            Ok(_) => {
                // Successfully read the expected number of bytes
                self.chk.crc = u32::from_be_bytes(crc_bytes);
            }
            Err(err) if err.kind() == ErrorKind::UnexpectedEof => {
                // Handle the situation where the file ends before reading the expected byt
                eprintln!("Warning: Reached end of file prematurely while reading CRC");
            }
            Err(err) => {
                eprintln!("Error reading CRC bytes: {}", err);
            }
        }
    }

    fn chunk_type_to_string(&self) -> String {
        String::from_utf8_lossy(&self.chk.r#type.to_be_bytes()).to_string()
    }
}

let mut file = File::open("stegano/prj.png").expect("Error opening file");

let mut meta_chunk = MetaChunk::pre_process_image(&mut file).expect("Error processing image

meta_chunk.process_image(&mut file);
```

```
It is a valid PNG file. Let's process it!
---- Chunk # 1 ----
Chunk Offset: 8
---- Chunk # 2 ----
Chunk Offset: 21
---- Chunk # 3 ----
Chunk Offset: 2e
---- Chunk # 4 ----
Chunk Offset: 203a
---- Chunk # 5 ----
Chunk Offset: 4046
---- Chunk # 6 ----
Chunk Offset: 6052
---- Chunk # 7 ----
Chunk Offset: 805e
---- Chunk # 8 ----
Chunk Offset: a06a
---- Chunk # 9 ----
Chunk Offset: c076
---- Chunk # 10 ----
Chunk Offset: e082
---- Chunk # 11 ----
Chunk Offset: 1008e
---- Chunk # 12 ----
Chunk Offset: 1209a
---- Chunk # 13 ----
Chunk Offset: 140a6
---- Chunk # 14 ----
Chunk Offset: 160b2
---- Chunk # 15 ----
Chunk Offset: 180be
---- Chunk # 16 ----
Chunk Offset: 1a0ca
---- Chunk # 17 ----
Chunk Offset: 1c0d6
---- Chunk # 18 ----
Chunk Offset: 1e0e2
---- Chunk # 19 ----
Chunk Offset: 200ee
---- Chunk # 20 ----
Chunk Offset: 220fa
---- Chunk # 21 ----
Chunk Offset: 24106
---- Chunk # 22 ----
Chunk Offset: 26112
---- Chunk # 23 ----
```

```
Chunk Offset: 26d33
```


/home/mahmoud/Desktop/Rust Book Dark/dark-web-rust/chapter-2/stegano/prj.png - Bless
File Edit View Search Tools Help
prj.png

```
00026c9e AE 9C 53 45 80 16 14 4E C5 32 9A 21 0A 9D B6 18 A6 5B 70 ..SE...N.2.!.....[p
00026cb1 EA 58 B4 AB 1B 5A B1 4E 53 04 B6 0A 95 82 7E DB E2 D4 CC .X...Z.NS.....~....
00026cc4 41 53 C7 21 62 74 DC FA 7A F0 56 7D B5 73 38 32 57 73 BE AS.!bt..z.V}.s82Ws.
00026cd7 8F 1A 73 AD 79 1F C5 03 71 41 00 04 40 00 04 40 00 04 40 ..s.y...qA..@..@..@
00026cea 00 04 40 00 04 40 00 04 EE 6C 02 1A A1 F1 12 09 8D E4 DD ..@..@...l.........
00026cfd 9E 6B 17 25 DB 31 03 45 95 68 39 28 3C 8B B9 28 4B 52 D7 .k.%.1.E.h9(<..(KR.
00026d10 A1 E8 BA 6E 15 3B A5 EB 59 0D 35 3F C5 DA 5A A5 F3 AF DA ...n.;..Y.5?..Z....
00026d23 C7 AE 07 A7 BC 51 FF 3F 11 E1 A4 2C 24 35 B6 70 00 00 00 .....Q.?...,$5.p...
00026d36 00 49 45 4E 44 AE 42 60 82                               .IEND.B`.
```

Signed 8 bit: 0
Signed 32 bit: 0
Hexadecimal: 00 00 00 00
Unsigned 8 bit: 0
Unsigned 32 bit: 0
Decimal: 000 000 000 000
Signed 16 bit: 0
Float 32 bit: 0
Octal: 000 000 000 000
Unsigned 16 bit: 0
Float 64 bit: 6.07344418312165E-315
Binary: 00000000 00000000 0000(
Show little endian decoding
Show unsigned as hexadecimal
ASCII Text:
Offset: 0x26d33 / 0x26d3e
Selection: None
INS

Figure 3: ScreenShot taken from the Bless hex editor

The output signifies the successful validation of the PNG file as indicated by the confirmation message, “It is a valid PNG file. Let’s process it!” This message is printed after checking if the magic bytes “PNG” are present in the file.

Following this confirmation, the program proceeds to process the PNG file in a chunk-wise manner. Each line in the output corresponds to a different chunk within the PNG file, and the offset value indicates the position of each chunk in hexadecimal format. The term “Chunk Offset” is essentially revealing the starting point of each chunk within the file.

For instance, the first chunk (Chunk #1) starts at an offset of 8, the second chunk (Chunk #2) at an offset of 21, and so forth. These offsets provide a crucial reference point for navigating and manipulating the image file. In the context of steganography, understanding the position of each chunk becomes essential for hiding and extracting hidden data without corrupting the file structure.

Having meticulously traversed the header of the PNG image and comprehensively processed it in a chunk-wise fashion, we have gained invaluable insights into the complex structure of the image file. The journey has developed as a meticulous exploration of the PNG format, unraveling the magic bytes and delving into the sequential chunks that compose the image.

Now armed with the knowledge of how to navigate through the chunks and decipher their offsets, our next work is to embark on the hiding of a message within these layers of data. This entails the artful insertion of our payload into the file, ensuring its secret existence within the seemingly innocent image structure.

3. Hiding a secret in a PNG File

In the domain of steganography, where plenty of sophisticated techniques exist for embedding payloads, our focus within this particular section will sharpen on the meticulous method of manipulating byte offsets. The PNG file format, our chosen format for this exploration, introduces a structured canvas outlined by additional chunk segments, each carefully defined within its specification. These critical chunks form the ground upon which the image decoder relies for seamless image processing. On the other hand, the additional chunks, while optional, contribute essential metadata that extends beyond the core encoding or decoding processes, including timestamps and textual information.

Within this complex framework, the additional chunk type emerges as an opportune target for the strategic injection of payloads. Its non-essential role in the image processing pipeline makes it an ideal candidate for manipulation. Whether through overwriting an existing chunk or delicately introducing an entirely new one, the potential for impactful alteration is vast. This sets the stage for a detailed exploration into the technique of seamlessly inserting new byte slices into the carefully delineated boundaries of an additional chunk segment.

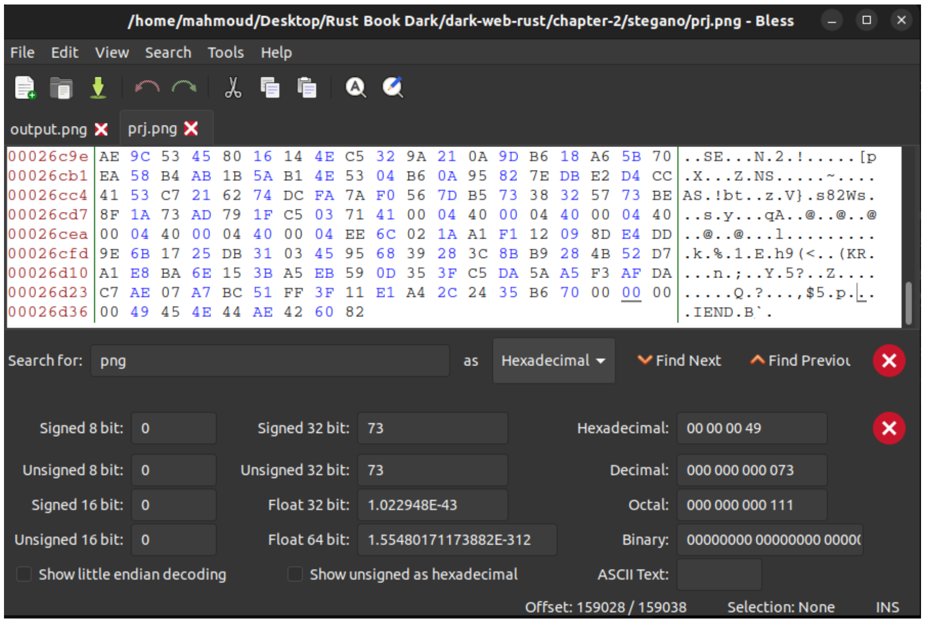
To truly grasp the complexities of this method, we must delve deep into the anatomy of the PNG file format, which was already discussed in the previous sections. The critical chunks, acting as the digital scaffolding of the image, house critical data for the image decoder's effective functioning. Contrariwise, the additional chunks, characterized by their optional nature, impart an additional layer of functionality to the file by contributing supplementary metadata. This metadata, while not necessary to the decoding process, adds intrinsic value by encapsulating timestamps and textual details.

The deliberate selection of the additional chunk type as the focal point for payload injection derives from its non-critical identification. By opting to overwrite an existing ancillary chunk or introduce a completely new one, the image can be manipulated without compromising its core decoding functionality. In the subsequent sections, we will guide you through a meticulously detailed, step-by-step process, shedding light on the sophisticated art of seamlessly integrating new byte slices into the carefully chosen chunk segment. This journey aims to unravel the complexities of steganographic payload embedding, offering a comprehensive understanding of the interplay between manipulation and the complex landscape of PNG file structures.

3.1 Locating an offset

To pinpoint an appropriate chunk offset, the initial step entails determining an appropriate location within the image data. Revisiting the hex editor, we proceed to walk through the original PNG file while moving toward the end of the image file.

Within any valid `PNG` image, an `IEND` chunk type serves as an indicator for the terminal section of the file, signifying the end-of-file (`EOF`) chunk. Precision in this process involves advancing to the 4 bytes immediately preceding the final `SIZE` chunk, strategically situating yourself at the initiation offset of the IEND chunk. This marks the beginning of the arbitrary chunks - whether they be critical or not - encompassed within the overarching PNG file. It's essential to acknowledge that these chunks are optional, making it plausible that the file under examination may not exhibit the same chunks or might lack them altogether. In our illustrative scenario, the offset leading to the IEND chunk originates at an integer offset **159020**, as shown in the following image.


/home/mahmoud/Desktop/Rust Book Dark/dark-web-rust/chapter-2/stegano/prj.png - Bless
File Edit View Search Tools Help
output.png
prj.png

```
00026c9e AE 9C 53 45 80 16 14 4E C5 32 9A 21 0A 9D B6 18 A6 5B 70 ..SE...N.2.!.....[p
00026cb1 EA 58 B4 AB 1B 5A B1 4E 53 04 B6 0A 95 82 7E DB E2 D4 CC .X...Z.NS.....~....
00026cc4 41 53 C7 21 62 74 DC FA 7A F0 56 7D B5 73 38 32 57 73 BE AS.!bt..z.V}.s82Ws.
00026cd7 8F 1A 73 AD 79 1F C5 03 71 41 00 04 40 00 04 40 00 04 40 ..s.y...qA..@..@..@
00026cea 00 04 40 00 04 40 00 04 EE 6C 02 1A A1 F1 12 09 8D E4 DD ..@..@...l.........
00026cfd 9E 6B 17 25 DB 31 03 45 95 68 39 28 3C 8B B9 28 4B 52 D7 .k.%.1.E.h9(<..(KR.
00026d10 A1 E8 BA 6E 15 3B A5 EB 59 0D 35 3F C5 DA 5A A5 F3 AF DA ...n.;..Y.5?..Z....
00026d23 C7 AE 07 A7 BC 51 FF 3F 11 E1 A4 2C 24 35 B6 70 00 00 00 .....Q.?...,$5.p...
00026d36 00 49 45 4E 44 AE 42 60 82                               .IEND.B`.
```

Search for: png as Hexadecimal

Signed 8 bit: 0 | Signed 32 bit: 73 | Hexadecimal: 00 00 00 49
Unsigned 8 bit: 0 | Unsigned 32 bit: 73 | Decimal: 000 000 000 073
Signed 16 bit: 0 | Float 32 bit: 1.022948E-43 | Octal: 000 000 000 111
Unsigned 16 bit: 0 | Float 64 bit: 1.55480171173882E-312 | Binary: 00000000 00000000 0000(
Show little endian decoding | Show unsigned as hexadecimal | ASCII Text:

Offset: 159028 / 159038 Selection: None INS

Figure 4: ScreenShot taken from the Bless hex editor

This image visually encapsulates the process of determining a chunk offset in relation to the `IEND` position. The complex interplay of the file's structure unfolds as you navigate through the hex representation, unraveling the spatial arrangement of critical chunks within the `PNG` file. This meticulous exploration ensures a comprehensive understanding of the file's composition, particularly when dealing with optional chunks that contribute to the overall complexity of the `PNG` format. In essence, this methodological approach establishes a foundation for precise offset identification, allowing for an examination of `PNG` files and facilitating informed analysis of their structural complexities.

3.2 Writing the payload

Navigating the complexities of encoding ordered bytes into a byte stream often involves utilizing a Rust struct as a standard approach. The following code snippet orchestrates a sequence of functions that are progressively developed throughout this section.

```rust
struct CmdArgs {
    input: String,
    output: String,
    meta: bool,
    suppress: bool,
    offset: String,
    inject: bool,
    payload: String,
    r#type: String,
    encode: bool,
    decode: bool,
    key: String,
}

impl CmdArgs {
    fn new(args: &[String]) -> Result<Self, &'static str> {
        if args.len() < 5 {
            return Err("Not enough arguments. Usage: program input output offset payload");
        }

        Ok(CmdArgs {
            input: args[1].clone(),
            output: args[2].clone(),
            meta: false,
            suppress: false,
            offset: args[3].clone(),
            inject: false,
            payload: args[4].clone(),
            r#type: String::from("PNG"),
            encode: args.contains(&String::from("encode")),
            decode: args.contains(&String::from("decode")),
            key: args[5].clone(),
        })
    }
}
```

The `CmdArgs` struct, as outlined in the above code snippet, encapsulates flag values obtained from command line inputs. These flags play a pivotal role in determining the payload to be utilized and the specific location within the image data for insertion. The struct employs these flags to create a new `MetaChunk` struct instance `meta_chunk`, aligning with the structured format of bytes to be written, mirroring those read from existing chunk segments.

```
fn main() {
    let args: Vec<String> = env::args().collect();

    let cmd_line_opts = match CmdArgs::new(&args) {
        Ok(opts) => opts,
        Err(err) => {
            eprintln!("Error: {}", err);
            return;
        }
    };

    let mut file = File::open(&cmd_line_opts.input).expect("Error opening file");

    let mut meta_chunk = MetaChunk::pre_process_image(&mut file).expect("Error processing i

    if cmd_line_opts.encode {
        let mut file_writer = File::create(&cmd_line_opts.output).unwrap();

        // Calculate CRC for the payload
        let mut bytes_msb = Vec::new();
        bytes_msb.write_all(&meta_chunk.chk.r#type.to_be_bytes()).unwrap();
        bytes_msb.write_all(&cmd_line_opts.payload.as_bytes()).unwrap();
        let crc = crc32_little(meta_chunk.chk.crc, &bytes_msb);

        // Update the MetaChunk with the encoded data and CRC
        meta_chunk.chk.data = encoded_data;
        meta_chunk.chk.crc = crc;

        // Create a new mutable reference to file_reader
        let mut file_reader = &file;

        meta_chunk.write_data(&mut file_reader, &cmd_line_opts, &mut file_writer);

        println!("Image encoded and written successfully!");
    } else if cmd_line_opts.decode {
        // TODO: Find and decode the payload
        meta_chunk.process_image(&mut file);
    }
}
```

Upon initializing the `MetaChunk` struct, the next procedural step involves reading the payload into a byte slice `file_writer`. However, this necessitates additional functionality to convert the literal flag values into a practical byte array. The following sections meticulously unravel the complexities of functions such as `marshal_data`, `write_data` contributing to the overall functionality.

The difficulties involved in computing the **CRC32** of the chunk mandate the innovative development of a new crate known as **`crc32-v2`**. This remarkable creation stands as the evolutionary successor to the pre-existing **`crc32`** crate, showing a new era of enhanced functionality and performance in the domain of **cyclic redundancy checks**. The birth of `crc32-v2` signifies a quantum leap in crate evolution, meticulously tailored to address the specific demands and challenges posed by the slightly complex task of CRC32 computation for the chunk. This crate, not only builds upon the foundation laid by its predecessor but also introduces novel methodologies and optimizations (TODO), marking a paradigm shift in the landscape of CRC32 computation in the domain of chunk processing. The crafting of `crc32-v2` is a testament to the relentless pursuit of excellence, where every line of code reflects a synthesis of brilliance and precision, elevating the standards of crate development to unprecedented heights.

```
impl MetaChunk {
    fn marshal_data(&self) -> Vec<u8> {
        let mut bytes_msb = Vec::new();
        bytes_msb.push(self.chk.data.len() as u8);
        bytes_msb.write_all(&self.chk.r#type.to_be_bytes()).unwrap();
        bytes_msb.write_all(&self.chk.data).unwrap();
        bytes_msb.write_all(&self.chk.crc.to_be_bytes()).unwrap();
        println!("Encoded Payload: {:?}", bytes_msb);
        bytes_msb
    }

    fn write_data<R: Read + Seek, W: Write>(&self, r: &mut R, c: &CmdArgs, mut w: W) {
        // write header at position 0
        let b_arr = u64_to_u8_array(self.header.header);
        w.write_all(&b_arr).unwrap();
        // write from 0 to offset
        let offset = i64::from_str(&c.offset).unwrap();
        let mut buff = vec![0; offset as usize];
        r.read_exact(&mut buff).unwrap();
        w.write_all(&buff).unwrap();
        // write payload at offset
        let data: Vec<u8> = self.marshal_data();
        w.write_all(&data).unwrap();
        // write from offset to end
        // uncomment the following line to preserve the length of the image after manipulat
        // r.seek(SeekFrom::Current(data.len().try_into().unwrap())).expect("Error seeking
        copy(r, &mut w).unwrap();
    }
}
```

Starting with the `marshal_data` method, as shown in the above code snippet, orchestrates the consolidation of all essential chunk information into a bytes buffer, incorporating size, type, data, and checksum. This method plays a pivotal role in preparing the raw bytes of the custom chunk for insertion into the new image file.

The final piece of the puzzle lies in the `write_data` function, the struct method as in the previous section. This function accepts a buffer reader `r`, `CmdArgs` struct `c`, and a buffer writer `w` facilitates the creation of a new PNG image by strategically writing the new chunk segment bytes into the specified offset location. The function adeptly handles nuances such as preserving the PNG header bytes, ensuring consistency between the original and modified images.

In essence, this complex sequence of methods and functions forms the backbone of a steganography program, seamlessly integrating ordered bytes into a byte stream and topping in the successful injection of a payload as a new chunk. Through careful execution of the command line program, as exemplified in the next section, a new PNG file, namely "output.png" is generated, preserving identical leading and trailing chunks while secretly hiding the injected payload.

This meticulous exploration is a sense of accomplishment, marking the successful creation of a steganography program that seamlessly integrates ordered bytes into a PNG file, displaying a robust foundation for further exploration and refinement in the realm of data hiding and security.

3.3 Encoding a payload using a simple XOR operation

In the realm of steganography, where various techniques are employed to obscure data within binary files, an **obfuscation** method plays a pivotal role. Building upon the groundwork laid in the previous sections, our focus now extends to enhancing the program with the incorporation of obfuscation techniques, thereby hiding the true value of the payload.

The utility of obfuscation becomes particularly pronounced in scenarios where concealing the payload from vigilant network-monitoring devices and robust endpoint security solutions is imperative. Consider a scenario where a raw **shellcode**, designed for spawning a new **Metasploit shell**, is being embedded. In such cases, evasion of detection becomes critical. To achieve this, the utilization of **Exclusive OR (XOR)** bitwise operations for both encryption and decryption of data emerges as a strategic choice.

XOR, a conditional operation between two binary values, yields a Boolean true outcome only when the two values differ. Contrariwise, a Boolean false result is produced if the values are identical. The XOR truth table, as outlined in the following table, briefly captures this logic, where the output is true when either x or y is true, but not when both are true.

x	y	x XOR y
0	0	0
0	1	1
1	0	1
1	1	0

This XOR logic forms the crux of data obfuscation, wherein the bits in the data are compared to the bits of a secret key. The subsequent manipulation involves altering the payload bit to 0 when the two values match and to 1 when they differ. To implement this concept, the code from the previous section is expanded to accommodate an `xor_encode_decode` function.

```rust
fn xor_encode_decode(input: &[u8], key: &str) -> Vec<u8> {
    let mut b_arr = Vec::with_capacity(input.len());
    for (i, &byte) in input.iter().enumerate() {
        b_arr.push(byte.wrapping_add(key.as_bytes()[i % key.len()]));
    }
    b_arr
}
```

The `xor_encode_decode` function, at its core, takes a byte slice representing the payload (`input`) and a secret key (`key`) as inputs. Within its scope, a new byte vector (`b_arr`) is initialized with a length equivalent to that of the input payload. Subsequently, a conditional loop iterates over each index position of the input byte array. During each iteration, XOR operations are performed by taking the current index's binary value and XORing it with a binary value derived from the modulo of the current index and the length of the secret key (`key`). This cleverly enables the use of a key shorter than the payload, as the modulo ensures that when the end of the key is reached, the next iteration reverts to the first byte of the key. The resulting XOR operation outputs are then written to the new `b_arr` bytes vector, which is ultimately returned.

Practical integration of these XOR functions into the existing program entails modifying the `main` function logic. The modifications, illustrated in the following code snippet, assume the use of command line arguments to dynamically pass values for conditional encoding and decoding logic.

```rust
fn main() {
    // ..
    if cmd_line_opts.encode {
        let mut file_writer = File::create(&cmd_line_opts.output).unwrap();
        // Assuming encoding is requested
        let encoded_data = xor_encode_decode(cmd_line_opts.payload.as_bytes(), &cmd_line_opt

        // Calculate CRC for the encoded data
        let mut bytes_msb = Vec::new();
        bytes_msb.write_all(&meta_chunk.chk.r#type.to_be_bytes()).unwrap();
        bytes_msb.write_all(&encoded_data).unwrap();
        let crc = crc32_little(meta_chunk.chk.crc, &bytes_msb);

        // Update the MetaChunk with the encoded data and CRC
        meta_chunk.chk.data = encoded_data;
        meta_chunk.chk.crc = crc;

        // Create a new mutable reference to file_reader
        let mut file_reader = &file;

        meta_chunk.write_data(&mut file_reader, &cmd_line_opts, &mut file_writer);

        println!("Image encoded and written successfully!");
    } else if cmd_line_opts.decode {
        // ..
    }
}
```

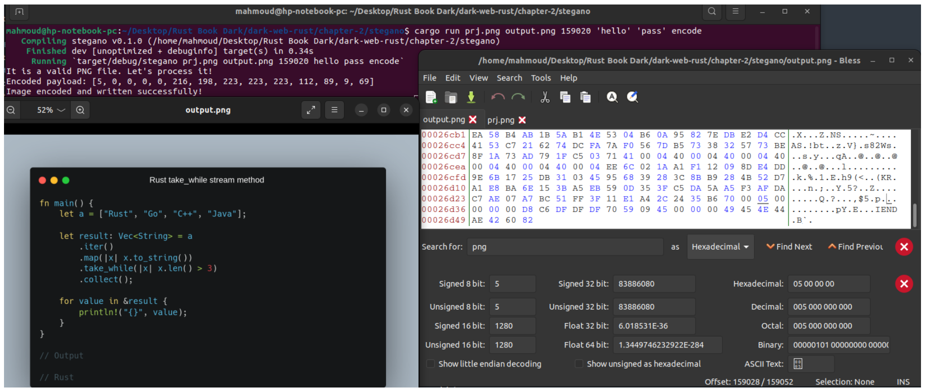
Now, let's put it all together.

3.4 Encode and Inject a payload program

```rust
use std::process::{Command, Output, Stdio};

// A helper function to execute a shell command from a Rust script
fn execute_command(command: &str) -> Result<(), std::io::Error> {
    let status = Command::new("bash")
        .arg("-c")
        .arg(command)
        .stderr(Stdio::inherit())
        .status()?;

    if status.success() {
        Ok(())
    } else {
        Err(std::io::Error::from_raw_os_error(status.code().unwrap_or(1)))
    }
}
```

```rust
let command = "cd stegano && cargo run prj.png output.png 159028 'hello' 'pass' encode";

if let Err(err) = execute_command(command) {
    eprintln!("Error executing command: {}", err);
}
```

```
    Finished dev [unoptimized + debuginfo] target(s) in 0.03s
     Running `target/debug/stegano prj.png output.png 159028 hello pass encode`


It is a valid PNG file. Let's process it!
original bytes [104, 101, 108, 108, 111]
Encoded Payload: [5, 0, 0, 0, 0, 24, 4, 31, 31, 31, 109, 44, 172, 118]
Image encoded and written successfully!




()
```



```
mahmoud@hp-notebook-pc:~/Desktop/Rust Book Dark/dark-web-rust/chapter-2/stegano$ cargo run prj.png output.png 159020 'hello' 'pass' encode
   Compiling stegano v0.1.0 (/home/mahmoud/Desktop/Rust Book Dark/dark-web-rust/chapter-2/stegano)
    Finished dev [unoptimized + debuginfo] target(s) in 0.34s
     Running `target/debug/stegano prj.png output.png 159020 hello pass encode`
It is a valid PNG file. Let's process it!
Encoded payload: [5, 0, 0, 0, 0, 216, 198, 223, 223, 223, 112, 89, 9, 69]
Image encoded and written successfully!
```

```rust
fn main() {
    let a = ["Rust", "Go", "C++", "Java"];

    let result: Vec<String> = a
        .iter()
        .map(|x| x.to_string())
        .take_while(|x| x.len() > 3)
        .collect();

    for value in &result {
        println!("{}", value);
    }
}

// Output

// Rust
```

```
00026cb1 EA 58 B4 AB 1B 5A B1 4E 53 04 B6 0A 95 82 7E DB E2 D4 CC .X...Z.NS.....~....
00026cc4 41 53 C7 21 62 74 DC FA 7A F0 56 7D B5 73 38 32 57 73 BE AS.!bt..z.V}.s82Ws.
00026cd7 8F 1A 73 AD 79 1F C5 03 71 41 00 04 40 00 04 40 00 04 40 ..s.y...qA..@..@..@
00026cea 00 04 40 00 04 40 00 04 EE 6C 02 1A A1 F1 12 09 8D E4 DD ..@..@...l.........
00026cfd 9E 6B 17 25 DB 31 03 45 95 68 39 28 3C 8B B9 28 4B 52 D7 .k.%.1.E.h9(<..(KR.
00026d10 A1 E8 BA 6E 15 3B A5 EB 59 0D 35 3F C5 DA 5A A5 F3 AF DA ...n.;..Y.5?..Z....
00026d23 C7 AE 07 A7 BC 51 FF 3F 11 E1 A4 2C 24 35 B6 70 00 05 00 .....Q.?...,$5.p...
00026d36 00 00 00 D8 C6 DF DF DF 70 59 09 45 00 00 00 49 45 4E 44 ........pY.E...IEND
00026d49 AE 42 60 82                                              .B`.
```

Search for: png as Hexadecimal

Signed 8 bit: 5 Signed 32 bit: 83886080 Hexadecimal: 05 00 00 00
Unsigned 8 bit: 5 Unsigned 32 bit: 83886080 Decimal: 005 000 000 000
Signed 16 bit: 1280 Float 32 bit: 6.018531E-36 Octal: 005 000 000 000
Unsigned 16 bit: 1280 Float 64 bit: 1.3449746232922E-284 Binary: 00000101 00000000 0000(
Show little endian decoding Show unsigned as hexadecimal ASCII Text:
Offset: 159028 / 159052 Selection: None INS

mahmoud@hp-notebook-pc: ~/Desktop/Rust Book Dark/dark-web-rust/chapter-2/stegano
52%
output.png
Rust take_while stream method
/home/mahmoud/Desktop/Rust Book Dark/dark-web-rust/chapter-2/stegano/output.png - Bless
File Edit View Search Tools Help
output.png
prj.png
Find Next
Find Previou

Figure 5: ScreenShot taken from the Bless hex editor

In essence, the integration of XOR-based obfuscation techniques into the program provides a robust layer of security, enabling the obscuring and protection of sensitive payloads within binary files. This strategic utilization of XOR operations serves as a powerful tool in the arsenal of data protection and steganographic practices, contributing to the multifaceted landscape of information security.

In the next sections, we are going to explore how to reverse these steps and extract the original payload.

```rust
let command = "cd stegano && cargo run output.png decoded.png 159020 'pass' decode";

if let Err(err) = execute_command(command) {
    eprintln!("Error executing command: {}", err);
}
```

---

4. Revealing a secret from a PNG File

Now that we were successfully able to encode and inject a secret inside a PNG file by pinpointing an offset and embedding information at that specific location, the subsequent phase involves the extraction of the secret information and the revelation of the hidden secrets encoded within, which is achieved through the decoding process utilizing the XOR operator. To do so, the user is required to specify not only the key employed in the XOR operation but also the image containing the concealed information and the precise offset where the secret was injected.

```
    fn write_data<R: Read + Seek, W: Write>(&mut self, r: &mut R, c: &CmdArgs, mut w: W) {
        // Common encoding and decoding process
        let b_arr = u64_to_u8_array(self.header.header);
        w.write_all(&b_arr).unwrap();
        let offset = i64::from_str(&c.offset).unwrap();
        let mut buff = vec![0; (offset - 8) as usize];

        if c.encode {
            // Encoding specific operations
            buff.resize((offset - 8) as usize, 0);
            r.read_exact(&mut buff).unwrap();
            w.write_all(&buff).unwrap();
            let data: Vec<u8> = self.marshal_data();
            w.write_all(&data).unwrap();
            // Uncomment the following line to preserve the length of the image after manip
            // r.seek(SeekFrom::Current(data.len().try_into().unwrap())).expect("Error seek
            copy(r, &mut w).unwrap();
        } else if c.decode {
            // Decoding specific operations
            buff.resize((offset - 16) as usize, 0);
            r.read_exact(&mut buff).unwrap();
            w.write_all(&buff).unwrap();
            let offset = self.get_offset(r);
            self.read_chunk(r);
            println!("Encoded Payload: {:?}", self.chk);
            let decoded_data = xor_encode_decode(&self.chk.data, &c.key);
            let decoded_string = String::from_utf8_lossy(&decoded_data);
            println!("Decoded Payload: {:?}", decoded_data);
            println!("Original Data: {:?}", decoded_string);
            r.seek(SeekFrom::Current(self.chk.data.len().try_into().unwrap()))
                .expect("Error seeking to offset");
            copy(r, &mut w).unwrap();
        }
    }
```

Now, let's break down the method responsible for this cryptographic work - the write_data method. This method is used to handle the dual responsibilities of injecting and encoding, as well as extracting and decoding the hidden information.

```rust
fn write_data<R: Read + Seek, W: Write>(&mut self, r: &mut R, c: &CmdArgs, mut w: W) {
    // ... (omitted for brevity)
```

At the method's entry point, we encounter generic parameters `R` and `W`, which represent types that can be read from (`Read`) and written to (`Write`). These generics make the method adaptable to various input and output sources.

```rust
if c.encode {
    // Encoding process
    let b_arr = u64_to_u8_array(self.header.header);
    w.write_all(&b_arr).unwrap();
```

The conditional statement checks if encoding is the goal. If so, the method initiates the encoding process by converting the header's 64-bit value into a byte array and writing it to the output stream (`w`).

```rust
let offset = i64::from_str(&c.offset).unwrap();
let mut buff = vec![0; (offset - 8) as usize];
r.read_exact(&mut buff).unwrap();
w.write_all(&buff).unwrap();
```

Here, the method reads data from the input stream (`r`) up to the specified offset, preparing the canvas for the payload insertion. It's like making space for the hidden message within the image.

```rust
let data: Vec<u8> = self.marshal_data();
w.write_all(&data).unwrap();
```

The payload, encapsulated as a vector of unsigned 8-bit integers, is then written to the output stream (`w`). Think of this step as the act of placing the secret message at the designated location within the image.

```rust
// Uncomment the following line to preserve the length of the image after manipulation
// r.seek(SeekFrom::Current(data.len().try_into().unwrap())).expect("Error seeking to offse
```

This commented line when uncommented, ensures that the length of the image remains intact after manipulation. It's like tidying up after the secret has been securely placed.

```rust
copy(r, &mut w).unwrap();
```

Finally, the method completes the encoding process by copying the remaining data from the input stream (`r`) to the output stream (`w`). This step ensures that the image remains coherent even after the injection operation.

```rust
} else if c.decode {
    // Decoding process
    let b_arr = u64_to_u8_array(self.header.header);
    w.write_all(&b_arr).unwrap();
```

The method then delves into the decoding process if the user specifies the `decode` flag. Like before, the header is written to the output stream (`w`), setting the stage for the revelation of hidden information.

```
let offset = i64::from_str(&c.offset).unwrap();
let mut buff = vec![0; (offset - 16) as usize];
r.read_exact(&mut buff).unwrap();
w.write_all(&buff).unwrap();
```

Similarly to the encoding process, data up to the specified offset is read from the input stream (`r`). The buffer (`buff`) ensures that the canvas is prepared for decoding.

```
let offset = self.get_offset(r);
self.read_chunk(r);
println!("Encoded Payload: {:?}", self.chk);
```

The method then fetches the actual offset of the hidden payload, reads the payload chunk, and prints the encoded payload for the user to witness the encoded information.

```
let decoded_data = xor_encode_decode(&self.chk.data, &c.key);
let decoded_string = String::from_utf8_lossy(&decoded_data);
println!("Decoded Payload: {:?}", decoded_data);
println!("Original Data: {:?}", decoded_string);
```

Using the XOR operator and the provided key, the encoded payload is decoded. The resulting decoded data is printed for the user, revealing the hidden secret within the image.

```
r.seek(SeekFrom::Current(self.chk.data.len().try_into().unwrap())).expect("Error seeking to
copy(r, &mut w).unwrap();
```

To conclude the decoding process, the method seeks to the position following the decoded payload's length within the input stream (`r`). The remaining data is then copied to the output stream (`w`), ensuring the image's integrity is maintained.

4.1 Extract and Decode the secret program

```
let command = "cd stegano && cargo run output.png decoded.png 159028 decode 'pass'";

if let Err(err) = execute_command(command) {
    eprintln!("Error executing command: {}", err);
}
```

```
    Finished dev [unoptimized + debuginfo] target(s) in 0.04s
     Running `target/debug/stegano output.png decoded.png 159028 decode pass`


It is a valid PNG file. Let's process it!
Encoded Payload: Chunk { size: 5, type: 0, data: [24, 4, 31, 31, 31], crc: 1831644278 }
Decoded Payload: [104, 101, 108, 108, 111]
Original Data: "hello"






()
```

As you can see, we've effectively managed to extract and decode the hidden message embedded within the picture. In simpler terms, we've revealed and translated the secret content that was previously hidden.

---

5. Test Cases

5.1 Exact Key, Exact Offset

```rust
let command = "cd stegano && cargo run prj.png output.png 159028 'hello there 121321412 1231

if let Err(err) = execute_command(command) {
    eprintln!("Error executing command: {}", err);
}
```

```
    Finished dev [unoptimized + debuginfo] target(s) in 0.03s
     Running `target/debug/stegano prj.png output.png 159028 'hello there 121321412 12312942


It is a valid PNG file. Let's process it!
original bytes [104, 101, 108, 108, 111, 32, 116, 104, 101, 114, 101, 32, 49, 50, 49, 51, 50
Encoded Payload: [30, 0, 0, 0, 0, 24, 4, 31, 31, 31, 65, 7, 27, 21, 19, 22, 83, 65, 83, 66,
Image encoded and written successfully!






()
```

```rust
let command = "cd stegano && cargo run output.png decoded.png 159028 decode 'pass'";
```

```
if let Err(err) = execute_command(command) {
    eprintln!("Error executing command: {}", err);
}
```

```
    Finished dev [unoptimized + debuginfo] target(s) in 0.05s
     Running `target/debug/stegano output.png decoded.png 159028 decode pass`


It is a valid PNG file. Let's process it!
Encoded Payload: Chunk { size: 30, type: 0, data: [24, 4, 31, 31, 31, 65, 7, 27, 21, 19, 22,
Decoded Payload: [104, 101, 108, 108, 111, 32, 116, 104, 101, 114, 101, 32, 49, 50, 49, 51,
Original Data: "hello there 121321412 12312942"
```

```
()
```

5.2 Wrong Key, Exact Offset

```
let command = "cd stegano && cargo run output.png decoded.png 159028 decode 'invalid-passwor

if let Err(err) = execute_command(command) {
    eprintln!("Error executing command: {}", err);
}
```

```
    Finished dev [unoptimized + debuginfo] target(s) in 0.05s
     Running `target/debug/stegano output.png decoded.png 159028 decode invalid-password`


It is a valid PNG file. Let's process it!
Encoded Payload: Chunk { size: 30, type: 0, data: [24, 4, 31, 31, 31, 65, 7, 27, 21, 19, 22,
Decoded Payload: [113, 106, 105, 126, 115, 40, 99, 54, 101, 114, 101, 32, 54, 60, 48, 36, 43
Original Data: "qji~s(c6ere 6<0$+>1#.(&l31293<"
```

```
()
```

5.3 Exact Key, Wrong Offset

```rust
let command = "cd stegano && cargo run output.png decoded.png 159000 decode 'pass'";

if let Err(err) = execute_command(command) {
    eprintln!("Error executing command: {}", err);
}
```

```
    Finished dev [unoptimized + debuginfo] target(s) in 0.03s
     Running `target/debug/stegano output.png decoded.png 159000 decode pass`


It is a valid PNG file. Let's process it!


Error reading chunk bytes: Reached end of file prematurely
Warning: Reached end of file prematurely while reading CRC


Encoded Payload: Chunk { size: 89, type: 221593541, data: [218, 90, 165, 243, 175, 218, 199,
Decoded Payload: [170, 59, 214, 128, 223, 187, 180, 221, 119, 198, 207, 34, 143, 94, 98, 14
Original Data: "?;n\u{7fb}??w??\"?^b??MWF?\u{11}smpasshello there 121321412 123129421?\u{b}





()
```

5.4 Wrong Key, Wrong Offset

```rust
let command = "cd stegano && cargo run output.png decoded.png 159000 decode 'invalid'";

if let Err(err) = execute_command(command) {
    eprintln!("Error executing command: {}", err);
}
```

```
    Finished dev [unoptimized + debuginfo] target(s) in 0.05s
     Running `target/debug/stegano output.png decoded.png 159000 decode invalid`


It is a valid PNG file. Let's process it!


Error reading chunk bytes: Reached end of file prematurely
Warning: Reached end of file prematurely while reading CRC
```

```
Encoded Payload: Chunk { size: 89, type: 221593541, data: [218, 90, 165, 243, 175, 218, 199,
Decoded Payload: [179, 52, 211, 146, 195, 179, 163, 199, 105, 209, 221, 61, 150, 91, 120, 14
Original Data: "?4Aó??i??=?[x??MH\\?\u{19}nhalidqji~s(cr{ew?(7+.41++&(,7\"<(.-=i?\u{17}]din?
```

()