

Forum:[Hack Forums](#)

Thread: [Steganography for beginners: hide a message in PNG pixels](#)

Dotexe - October 14, 2026, 4:11 am

? loading encrypted image...

Hello everyone, it's been quite a while since I last posted anything here, but I'm planning to become more active again. Over the years, I've made some great connections and gained a tremendous amount of knowledge from this community, so I figured it's about time I gave something back and contributed more myself. I plan to post a series of hacking tutorials in the next couple of days. Let's go!

Steganography for beginners: hide a message in PNG pixels

? loading encrypted image...

Introduction

A picture can carry a message that its viewer never sees as text. In this lesson, you will follow a short greeting into the color values of a generated image, save that image as PNG, and recover the greeting from the reopened file.

You need Python, the Pillow dependency installed, and enough familiarity with functions to follow their inputs and return values. A byte is a unit of eight bits, a bit is either zero or one. We will use both terms throughout. The program creates its own cover image and temporary file, so you do not need a photograph, account, network connection, or private message. Use the supplied fake greeting.

Steganography or how most people call it - Stegano means concealing data inside other data. This particular technique hides where the greeting is stored, but does not require a secret to retrieve it. That limitation is visible in the code, not something to discover after trusting it with confidential information.

? loading encrypted image...

The cover image and the message are separate inputs

```
Code cover = Image.new("RGB", (64, 64), (120,150, 180))
```

```
message = "Hello, beginner!".encode("utf-8")
```

```
stego = hide(cover, message)
```

The cover is the image that receives the hidden data. The message is a sequence of bytes. The resulting image, called the stego image, contains modified color values from which our extractor can reconstruct that sequence.

Our cover is a uniform RGB image. RGB means red, green and blue. Each pixel contributes a value for each of those channels. Pillow (our Python Image Library) supplies the conversion and pixel-buffer operations. The program does not append a text file to a PNG or use an image comment field. Its payload is carried by the pixel samples themselves.

Anyone with this extractor can read a matching payload. There is no password prompt, secret seed, or key argument. The exercise therefore answers how bytes can be concealed in a carrier, not how to authorize a recipient. Do not mistake a missing text label in an image viewer for an access-control rule.

? loading encrypted image...

From a greeting to a predictable pixel buffer

```
Code message = "Hello, beginner!".encode("utf-8")
```

```
stego = hide(cover, message)
```

The main block encodes Hello, beginner! as UTF-8 before calling hide(). In this example, the encoded message occupies 16 bytes. The visible text and its encoded bytes happen to have matching lengths because this greeting contains only ASCII characters.

For comparison, the text Dotexe contains six characters, and its UTF-8 representation occupies six bytes: 44 6f 74 65 78 65 in hexadecimal. This is a calculated example, not a second message printed by our main program.

```
Code def hide(image, message):
```

```
    rgb = image.convert("RGB")
```

```
    raw = bytearray(rgb.tobytes())
```

hide() converts the cover to RGB, calls tobytes(), and puts those bytes in a mutable bytearray. Pillow's tobytes() returns raw image data rather than the compressed contents of an image file. For the generated 64 by 64 cover, the RGB buffer contains 12,288 channel bytes. Both hide() and extract() use this representation, so their traversal agrees. The PNG file offsets and compressed file size have no role in the embedding calculation.

? loading encrypted image...

What replacing the least significant bit changes

```
Code raw[i] = (raw[i] & 0xFE) | bit
```

The least significant bit is the rightmost bit of a byte's binary representation. It determines whether the value is even or odd. Clearing it and inserting a new bit changes only that final position, the rest of the byte stays intact. Our implementation performs those operations with a bitwise AND followed by a bitwise OR.

The mask 0xFE has the binary representation 11111110. AND with that mask clears the final bit. OR with the chosen payload bit then inserts zero or one. A channel value of 120 becomes 121 when the chose bit is one. A value of 121 becomes 120 when the chose bit is zero. If the original final bit already matches, the channel does not change. Each affected channel therefore differs by at most one from its original value.

That bound applies to the RGB buffer used for embedding. It does not say that converting an arbitrary source image to RGB preserves everything about the source. Nor does a small numerical change prove invisibility. The generated cover is useful precisely because its simple baseline lets us reason about every change without making claims about photographs or detection software.

? loading encrypted image...

A small header gives the extractor a stopping point

```
Code payload = MAGIC + len(message).to_bytes(4, "big") + message
```

The payload is not just the greeting. It starts with the four byte marker STEG, continues with a

four-byte length field, and ends with the message bytes. A header is information that helps a reader interpret the following data. Here, the marker identifies our tutorial layout, and the length specifies how many message bytes to return.

```
Code header = read_bytes(8)
    if header[:4] != MAGIC:
        raise ValueError("No tutorial payload found")
    length = int.from_bytes(header[4:], "big")
    return read_bytes(8 + length)[8:]
```

For the unchanged greeting, the header in hexadecimal is 53 54 45 47 00 00 00 10. The first four bytes represent STEG. The remaining bytes encode the message length in big-endian order, meaning the most significant byte comes first. The writer and reader explicitly select the same order.

A terminator-based design would need rules for messages containing the terminator. This length-based design can carry arbitrary bytes, including zero bytes, without searching for a special ending character. It also represents an empty message, the marker and length remain, while the body has no bytes. However a recognizable marker only establishes that the format looks plausible. Somebody who knows the layout can construct the same marker and any message. There is no authenticity check attached to the header.

? loading encrypted image...
Follow the first payload byte into the image

```
Code for i in range(len(payload) * 8):
    bit = (payload[i // 8] >> (7 - i % 8)) & 1
```

The writer takes payload bits from left to right within each byte. The first payload byte represents S and has the calculated binary value 01010011. Its first bit is written into the first channel byte, its second into the next channel byte and so on. One payload byte consumes eight channel bytes.

For our initial color of red 120, green 150, and blue 180, the first three payload bits are zero, one, and zero. Applying the writer's expression makes the first pixel red 120, green 151, and blue 180. This follows from the actual helper, and illustrates how the payload crosses channel and pixel boundaries without treating a pixel as a message character.

```
Code for offset in range(0, count * 8, 8):
    value = 0
    for bit in raw[offset:offset + 8]:
        value = (value << 1) | (bit & 1)
    result.append(value)
return bytes(result)
```

The extractor takes each channel's final bit and reconstructs a value by shifting left before inserting the next bit. This preserves the same most significant bit first order used by the writer. Reading the same positions in the opposite order would produce different bytes, even though every individual bit was recovered correctly. Agreement about ordering is part of the format.

? loading encrypted image...
Capacity includes the header, not just the text

```
Code payload = MAGIC + len(message).to_bytes(4, "big") + message
    if len(payload) * 8 > len(raw):
        raise ValueError("Message does not fit")
```

The writer checks the entire payload before changing any channel. Every payload byte needs eight

available channel bytes, so the condition compares payload length multiplied by eight with the RGB buffer length. Equality is allowed though. An oversized message raises ValueError instead of silently cutting off the end.

```
Code print("Payload capacity:", len(cover.tobytes()) // 8 - 8, "bytes")
```

For this cover, 12,288 channel bytes provide room for 1,536 complete payload bytes. Subtracting the eight-byte header leaves 1,528 message bytes. These values were calculated from the actual dimensions and layout. A PNG with a larger compressed file size is not necessarily a higher capacity carrier, the expression uses decoded RGB dimensions.

The boundary test uses an 8 by 8 RGB image, whose message capacity is 16 bytes. It successfully hides 16 copies of x and rejects 17. If you choose a cover too small for the header, even an empty message cannot fit. Empty content still needs a structure that tells the extractor it is intentionally empty rather than absent.

? loading encrypted image...

Why the disk round trip matters

Code with TemporaryDirectory() as folder:

```
path = Path(folder) / "stego.png"
stego.save(path, format="PNG")
with Image.open(path) as reopened:
    recovered = extract(reopened)
```

PNG supports lossless storage of raster images. That makes it suitable for this experiment's exact pixel value round trip. We save the image containing the modified values, reopen that saved image, and recover the greeting from the reopened object. An in memory round trip alone would not exercise the save and load operations.

Lossless storage preserves the image values being stored, it does not prevent an editor from changing them first. A resized or otherwise altered image may still be saved as a perfectly valid PNG while carrying different final bits. The filename extension cannot certify preservation of our payload.

The program explicitly saves with format PNG rather than leaving the choice implicit. Pillow handles constructing, saving and reopening the image. The temporary directory keeps the artifact local and removes it after the block finishes.

? loading encrypted image...

Preparing our environment

Create a new folder on your machine and make sure that Python 3, venv and pip are already installed and available. We should create an isolated environment:

```
Code python3 -m venv .venv
source .venv/bin/activate
python -m pip install Pillow==12.3.0
Save the code as stegano.py.
```

```
Code from pathlib import Path
from tempfile import TemporaryDirectory
from PIL import Image
```

```
MAGIC = b"STEG"
```

```
def hide(image, message):
    rgb = image.convert("RGB")
    raw = bytearray(rgb.tobytes())
    payload = MAGIC + len(message).to_bytes(4, "big") + message
    if len(payload) * 8 > len(raw):
        raise ValueError("Message does not fit")
    for i in range(len(payload) * 8):
        bit = (payload[i // 8] >> (7 - i % 8)) & 1
        raw[i] = (raw[i] & 0xFE) | bit
    return Image.frombytes("RGB", rgb.size, bytes(raw))
```

```
def extract(image):
    raw = image.convert("RGB").tobytes()
    def read_bytes(count):
        if count * 8 > len(raw):
            raise ValueError("Truncated or invalid payload")
        result = bytearray()
        for offset in range(0, count * 8, 8):
            value = 0
            for bit in raw[offset:offset + 8]:
                value = (value << 1) | (bit & 1)
            result.append(value)
        return bytes(result)
    header = read_bytes(8)
    if header[:4] != MAGIC:
        raise ValueError("No tutorial payload found")
    length = int.from_bytes(header[4:], "big")
    return read_bytes(8 + length)[8:]
```

```
if __name__ == "__main__":
    cover = Image.new("RGB", (64, 64), (120, 150, 180))
    message = "Hello, beginner!".encode("utf-8")
    stego = hide(cover, message)
    with TemporaryDirectory() as folder:
        path = Path(folder) / "stego.png"
        stego.save(path, format="PNG")
        with Image.open(path) as reopened:
            recovered = extract(reopened)
    assert recovered == message
    print("Recovered:", recovered.decode("utf-8"))
    print("Payload capacity:", len(cover.tobytes()) // 8 - 8, "bytes")
    print("PNG round-trip: PASS")
```

? loading encrypted image...

Following the code

Code MAGIC = b"STEG"

```
def hide(image, message):
    rgb = image.convert("RGB")
    raw = bytearray(rgb.tobytes())
    payload = MAGIC + len(message).to_bytes(4, "big") + message
    if len(payload) * 8 > len(raw):
```

```
raise ValueError("Message does not fit")
```

The imports supply filesystem paths, a temp directory and Pillow's Image API. MAGIC is the public byte marker shared by the two helpers. `hide` func first normalizes the image to RGB and copies its raw data into a bytearray so individual channel values can be replaced. Concatenating MAGIC, `len(message).to_bytes(4, 'big')`, and `message` builds the complete payload. The capacity branch runs before the embedding loop so insufficient space never returns a partially written image.

```
Code for i in range(len(payload) * 8):
    bit = (payload[i // 8] >> (7 - i % 8)) & 1
    raw[i] = (raw[i] & 0xFE) | bit
    return Image.frombytes("RGB", rgb.size, bytes(raw))
```

Inside that loop, integer division by eight selects the current payload byte. The remainder identifies a position inside that byte and subtracting it from seven makes the writer begin with the most significant bit. Right shifting brings that bit to the final position AND with one discards the other bits. Clearing the current channel's final bit with 0xFE and OR-ing the selected bit writes it. `Image.frombytes` function constructs a new RGB image with the original dimensions from the modified buffer and the original cover is not overwritten.

```
Code def extract(image):
    raw = image.convert("RGB").tobytes()
    def read_bytes(count):
        if count * 8 > len(raw):
            raise ValueError("Truncated or invalid payload")
        result = bytearray()
        for offset in range(0, count * 8, 8):
            value = 0
            for bit in raw[offset:offset + 8]:
                value = (value << 1) | (bit & 1)
            result.append(value)
        return bytes(result)
```

this function obtains the matching RGB representation. Its nested read bytes count always reads from the start of the buffer. It first refuses requests needing more channel bytes than exist. For each group of eight channels, it initializes value to zero, shifts the accumulating value left, and inserts the current channel's final bit. The reconstructed bytes accumulate in a bytearray, then become immutable bytes meaning it can't be changed.

```
Code header = read_bytes(8)
    if header[:4] != MAGIC:
        raise ValueError("No tutorial payload found")
    length = int.from_bytes(header[4:], "big")
    return read_bytes(8 + length)[8:]
```

The first call reads the eight byte header. A marker mismatch raises `No Tutorial payload found` and `int from bytes` function interprets the remaining header bytes as a big endian length. The second call rereads the header and requested body together, the return expression then discards the header. Rereading is simple though not the most efficient possible design.

and finally our main block

```
Code if __name__ == "__main__":
    cover = Image.new("RGB", (64, 64), (120, 150, 180))
    message = "Hello, beginner!".encode("utf-8")
    stego = hide(cover, message)
    with TemporaryDirectory() as folder:
```

```
path = Path(folder) / "stego.png"
stego.save(path, format="PNG")
with Image.open(path) as reopened:
    recovered = extract(reopened)
assert recovered == message
```

The main block creates the cover, encodes the greeting and calls hide function within temporarydirectory function, it writes stego.png and extracts from image.open function while that file context is active. After cleanup, the assertion compares recovered and original bytes. The print statements show decoded text, computed capacity and the success label.

Run the code by python stegano.py and observe the results

Code Recovered: Hello, beginner!

Payload capacity: 1528 bytes

PNG round-trip: PASS

```
Code print("Recovered:", recovered.decode("utf-8"))
```

```
print("Payload capacity:", len(cover.tobytes()) // 8 - 8, "bytes")
```

```
print("PNG round-trip: PASS")
```

The recorded successful run recovered Hello, beginner!, reported 1528 bytes of message capacity, and printed PNG round-trip: PASS. A fresh local execution of the unchanged script produced the same output.

The accompanying test module exercises a Unicode PNG round trip and the maximum channel change, exact-capacity acceptance and oversized rejection, missing-marker rejection, empty-message recovery, and an image too small for a header. It does not explicitly forge a valid marker with an impossible length, measure detectability, or test image-host processing.

Separate local failure probes returned Message does not fit for the oversized boundary, No tutorial payload found for a blank cover, and Truncated or invalid payload for a one-pixel image. Those are intentionally invalid inputs, not failures of the recorded main demonstration. The truncated-data check also exists for a body length that exceeds available data, but that branch coverage should not be inferred from the tests listed above.

? loading encrypted image...

That wraps up this tutorial :)

Quick note: Do not attempt to transfer sensitive company data throughout this method, advanced EDR such as CrowdStrike will detect the behavior and this will require complex obfuscation approach to workaround that.

CL1B - October 14, 2026, 4:26 am

Thank you for this great read and intro to steganography.

I know a lot of junior mdevs get very excited over this ;)

Dotexe - October 14, 2026, 4:31 am

(Oct 14, 2026 04:26 AM)CL1B Wrote: [\[CITE\]](#) Thank you for this great read and intro to steganography.

I know a lot of junior mdevs get very excited over this ;)

Thank you for the kind words! It's definitely a fascinating topic, and there's always something new to explore the deeper you dive into the technical aspects. I've actually been considering continuing the

series with Cryptology, but this time starting from the absolute basics to make it more accessible to beginners, as some of the topics covered so far might be a bit overwhelming for those just getting started.

Stanley - *October 15, 2026, 12:35 pm*

I'm Stanley HF's helper bot. The OP (original poster) of this thread wants everyone to see this page and has elected to bump it to the top of the showthread listing. Since HF doesn't want a ton of LQP (Low Quality Posts) this is the best method of bumping your content. There is a small ?yte fee for each bump. Any OP can bump any of their threads on HF from the linked icon on the top of their first post.

If you haven't already read the first post of this thread we encourage you to do so.

Thanks,
Stanley

Bastille - *October 15, 2026, 10:16 pm*

That's the kind of content I like to see on HF, I have few tutorials ideas I'll work on, I could even use Infamous GFX for them.

Thanks for the share, interesting topic, didn't read all of it yet, but planning to continue

Dotexe - *October 16, 2026, 12:55 am*

(17 hours ago)Bastille Wrote: [\[CITE\]](#) That's the kind of content I like to see on HF, I have few tutorials ideas I'll work on, I could even use Infamous GFX for them.

Thanks for the share, interesting topic, didn't read all of it yet, but planning to continue
Thank you Bastille ?. My next one will most likely cover the web app security. Can't wait to see yours ?

Th3HulK - *October 16, 2026, 3:58 am*

Thanks for sharing this, HQ Content