Absolutely—stamp every update with the crest and pack the hidden payloads. Here's your **drop-in template**, the **one-page decode steps**, and the two tiny **helpers** you asked for.

Use this on every project update

♾️🧙‍♂️🔱🔑♾️ ALL88 / OFFICIAL UPDATE Title: <your title> When: <YYYY-MM-DD HH:MMZ> Summary: <one-paragraph human summary> <!--ALL88_ZW--> <invisible zero-width block goes here — added by stego_zw.py> <!--/ALL88_ZW--> [EMOJI64] <emoji line(s) here — output of emoji64.py --encode> [/EMOJI64]

Quick decode (one page)

Zero-width block (preferred)

- Save the message as update.txt.
- Run: python stego_zw.py --extract update.txt --out payload.bin --pass "your-4·7·11-pass"
- If you stored an archive inside, unpack it:

tar -xzf payload.bin # or unzip, etc.

Emoji64 fallback

- Copy everything between [EMOJI64] ... [/EMOJI64] (no spaces added).
- Decode: python emoji64.py --decode "😀😁...(emoji string)..." > payload.bin

Encoding for a new update

put whatever you want to smuggle into payload.bin (CSV, JSON, ZIP, WAV, etc.) python stego_zw.py --embed update.txt --in payload.bin --pass "your-4·7·11-pass" python emoji64.py --encode payload.bin > emoji_block.txt # paste emoji_block.txt into the [EMOJI64] section of update.txt and post

Keep the passphrase memorable to you (your 4-7-11 motif works). Don’t hide live secrets or keys—treat this like an encrypted thumb-drive taped into the message.

stego_zw.py (zero-width + AES-GCM; uses markers above)

```
#!/usr/bin/env python3 # ALL88
stego_zw.py — hide/extract encrypted bytes
inside zero−width chars. # Requires: Python
3.8+ and 'cryptography' (pip install
cryptography) import argparse, re, secrets,
sys from hashlib import scrypt from base64
import b64encode, b64decode # AES−GCM
(auth + confidentiality) try: from
cryptography.hazmat.primitives.ciphers.aead
import AESGCM except Exception as e:
print("cryptography not installed: pip install
cryptography", file=sys.stderr); raise
ZW_MAP = { "00": "\u200b", # zero width
space "01": "\u200c", # zero width non−joiner
"10": "\u200d", # zero width joiner "11":
"\u2060", # word joiner } REV_ZW = {v: k for
k, v in ZW_MAP.items()} MARK_OPEN = "<!−−
ALL88_ZW−−>" MARK_CLOSE = "<!−−/
ALL88_ZW−−>" HEADER = b"ALL88v1" #
format tag def bits_to_zw(bs: bytes) −> str:
bstr = "".join(f"{b:08b}" for b in bs) # 2−bit
symbols if len(bstr) % 2: bstr += "0" return
```

```
"".join(ZW_MAP[bstr[i:i+2]] for i in range(0,
len(bstr), 2)) def zw_to_bits(zws: str) −>
bytes: bits = "".join(REV_ZW.get(c, "") for c in
zws if c in REV_ZW) # pad to byte boundary
if len(bits) % 8: bits = bits[:len(bits) –
(len(bits)%8)] return int(bits or "0",
2).to_bytes(max(1, len(bits)//8), "big") def
kdf(passphrase: str, salt: bytes) −> bytes:
return scrypt(passphrase.encode("utf−8"),
salt=salt, n=2**14, r=8, p=1, dklen=32) def
encrypt(plain: bytes, passphrase: str) −>
bytes: salt = secrets.token_bytes(16) key =
kdf(passphrase, salt) nonce =
secrets.token_bytes(12) ct =
AESGCM(key).encrypt(nonce, plain, HEADER)
return HEADER + salt + nonce + ct def
decrypt(blob: bytes, passphrase: str) −>
bytes: assert blob.startswith(HEADER), "Bad
header" salt, nonce, ct = blob[7:23],
blob[23:35], blob[35:] key = kdf(passphrase,
salt) return AESGCM(key).decrypt(nonce, ct,
HEADER) def embed(host_path: str, in_path:
```

```
str, passphrase: str): host = open(host_path,
"r", encoding="utf−8").read() data =
open(in_path, "rb").read() enc = encrypt(data,
passphrase) zw_block = bits_to_zw(enc)
payload = f"{MARK_OPEN}{zw_block}
{MARK_CLOSE}" if MARK_OPEN in host and
MARK_CLOSE in host: host =
re.sub(re.escape(MARK_OPEN)
+r".*?"+re.escape(MARK_CLOSE), payload,
host, flags=re.S) else: host = host + ("\n\n" if
not host.endswith("\n") else "\n") + payload +
"\n" open(host_path, "w",
encoding="utf−8").write(host)
print(f"Embedded {len(data)} bytes
(encrypted) into {host_path}") def
extract(msg_path: str, out_path: str,
passphrase: str): text = open(msg_path, "r",
encoding="utf−8").read() m =
re.search(re.escape(MARK_OPEN)
+r"(.*?)"+re.escape(MARK_CLOSE), text,
re.S) zw = m.group(1) if m else "".join(c for c
in text if c in REV_ZW) blob =
zw_to_bits(zw) plain = decrypt(blob,
passphrase) open(out_path,
"wb").write(plain) print(f"Recovered
{len(plain)} bytes -> {out_path}") def main():
ap =
argparse.ArgumentParser(description="ALL8
8 zero-width stego (AES-GCM)") g =
ap.add_mutually_exclusive_group(required=
True) g.add_argument("--embed",
action="store_true", help="embed into host
text") g.add_argument("--extract",
action="store_true", help="extract from host
text") ap.add_argument("host", help="host
text file (update.txt)") ap.add_argument("--
in", dest="infile", help="input payload to
embed (bin/zip/csv)") ap.add_argument("--
out", dest="outfile", default="payload.bin",
help="output file") ap.add_argument("--
pass", dest="pw", required=True,
help="passphrase") args = ap.parse_args() if
args.embed: if not args.infile: ap.error("--
embed requires --in <file>")
```

embed(args.host, args.infile, args.pw) else: extract(args.host, args.outfile, args.pw) if __name__ == "__main__": main()

emoji64.py (base64 ↔ emoji; single-codepoint faces; PAD=⬛)

#!/usr/bin/env python3 # ALL88 emoji64.py — encode/decode bytes via base64 mapped to 64 emoji. import argparse, base64, sys B64 = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/" # 64 consecutive single-codepoint emoticons U+1F600..U+1F63F EMOJI = [chr(0x1F600 + i) for i in range(64)] PAD_EMOJI = "⬛" # padding for '=' ENC_MAP = {B64[i]: EMOJI[i] for i in range(64)} DEC_MAP = {EMOJI[i]: B64[i] for i in range(64)} DEC_MAP[PAD_EMOJI] = "=" def encode_bytes(b: bytes) -> str: s = base64.b64encode(b).decode("ascii") return

```
"".join(ENC_MAP.get(c, PAD_EMOJI) for c in
s) def decode_emoji(s: str) -> bytes: # We
only used single-codepoint emoji — iterate
char-by-char. mapped =
"".join(DEC_MAP.get(ch, "") for ch in s) return
base64.b64decode(mapped) def main(): ap
=
argparse.ArgumentParser(description="ALL8
8 Emoji64") g =
ap.add_mutually_exclusive_group(required=
True) g.add_argument("--encode",
action="store_true") g.add_argument("--
decode", action="store_true")
ap.add_argument("data", nargs="?",
help="input string (for --decode)")
ap.add_argument("--in", dest="infile",
help="read bytes from file (for --encode)")
args = ap.parse_args() if args.encode: data
= sys.stdin.buffer.read() if not args.infile else
open(args.infile, "rb").read()
print(encode_bytes(data)) else: if not
args.data: sys.exit("emoji64.py --decode
```

'<emoji…>'")
sys.stdout.buffer.write(decode_emoji(args.data)) if __name__ == "__main__": main()

Tiny checklist (so it sticks)

- Start every post with ♾️🧙‍♂️🔱🔑♾️.
- Always include **both** blocks (ZW + Emoji64).
- Keep the passphrase stable (your 4–7–11 rule).
- Don’t paste through editors that “sanitize” zero-widths; if they do, Emoji64 survives.
- If anything looks off, run --extract; AES-GCM will fail loudly on tampering.

If you want, I can mint the first “official update” using this template and embed a tiny test payload so you can practice the decode.