Password Strength Analyzer + Custom Wordlist Generator

Summary & Complete Script (pwtool.py)

Topic:

Password Strength Analyzer + Custom Wordlist Generator

Project Summary:

This project demonstrates a Python-based cybersecurity tool that analyzes password strength and generates personalized wordlists for educational and ethical security use. It includes both a Command Line Interface (CLI) and a simple Graphical User Interface (GUI) using Tkinter. The tool evaluates password entropy and pattern strength while helping users understand password security concepts.

Core Features:

• Password strength analysis using entropy or zxcvbn (if installed).
• Custom wordlist generation from user inputs such as names, pets, birth years, and custom keywords.
• Leetspeak variants, prefixes, suffixes, and year patterns are automatically applied.
• Exports results as ready-to-use text files for password auditing tools (Hashcat, Hydra, etc.).
• Simple GUI using Tkinter and complete CLI-based automation.
• Educational use for password awareness and secure credential creation.

Usage Examples:

```
# Analyze password strength
python pwtool.py --analyze "P@ssw0rd2021!"

# Generate a custom wordlist
python pwtool.py --generate --name "Alice Bob" --pet "Rex" --birth "1990" --years 2005-2024 -o mylist.txt --max 10000

# Launch GUI
python pwtool.py --gui
```

Installation (Optional Libraries):

```
pip install zxcvbn nltk
```

Full Python Script — pwtool.py

```
#!/usr/bin/env python3
"""
pwtool.py
Password Strength Analyzer + Custom Wordlist Generator
(Enhanced: interactive mode, clearer help, keyboard patterns, hashcat rules, zip export)

Usage (short):
    python pwtool.py --analyze "P@ssw0rd2021!"
    python pwtool.py --generate --name "Alice Bob" --pet "Rex" --birth "1990" --years 2005-2024 -o mylist.txt --max 10000
    python pwtool.py --gui
    python pwtool.py --interactive
    python pwtool.py --zip bundle.zip --include-sample

Notes:
 - This tool is intended for defensive testing, research, and education only.
 - Do not use on targets you don't have permission to test.
 - For best password analysis install zxcvbn (pip install zxcvbn).
"""

import argparse
import itertools
import math
import os
import re
import sys
import zipfile
from datetime import datetime
from pathlib import Path
from typing import List

# Optional imports
try:
    from zxcvbn import zxcvbn  # type: ignore
    HAVE_ZXCVBN = True
except Exception:
    HAVE_ZXCVBN = False

try:
    import nltk  # optional, not required by current code
    HAVE_NLTK = True
except Exception:
    HAVE_NLTK = False

# --- Config / common lists ---
COMMON_SUFFIXES = ["!", "123", "2020", "2021", "@", "#", "007", "321", "!!"]
COMMON_PREFIXES = ["", "the", "my", "i", "x", "_"]
COMMON_WORDS = ["password", "admin", "welcome", "letmein", "qwerty", "football", "lovely"]
LEET_MAP = {
    "a": ["4", "@"],
    "b": ["8"],
    "e": ["3"],
    "i": ["1", "!"],
    "l": ["1", "7"],
    "o": ["0"],
    "s": ["5", "$"],
    "t": ["7"],
    "g": ["9"],
}

# keyboard adjacency sequences (short examples)
KEYBOARD_PATTERNS = [
    "qwertyuiop", "asdfghjkl", "zxcvbnm",
    "1234567890", "0987654321",
    "!@#$%^&*()", "qazwsx", "1q2w3e"
]

# --- Utility functions ---

def estimate_entropy(password: str) -> float:
    if not password:
        return 0.0
    pool = 0
    if re.search(r"[a-z]", password): pool += 26
    if re.search(r"[A-Z]", password): pool += 26
    if re.search(r"[0-9]", password): pool += 10
    if re.search(r"[^\w\s]", password): pool += 32
    pool = max(pool, 1)
    base_entropy = len(password) * math.log2(pool)
```

```python
    penalty = 0.0
    if re.fullmatch(r"(.)\1+$", password):
        penalty += 10
    low = password.lower()
    for w in COMMON_WORDS:
        if w in low:
            penalty += min(6 + len(w), 20)
    if re.search(r"(19|20)\d\d", password):
        penalty += 6
    if len(password) < 8:
        penalty += (8 - len(password)) * 2

    entropy = max(base_entropy - penalty, 0.0)
    return entropy

def entropy_score_label(entropy: float) -> str:
    if entropy < 28: return "Very weak"
    if entropy < 36: return "Weak"
    if entropy < 60: return "Reasonable"
    if entropy < 80: return "Strong"
    return "Very strong"

def analyze_password(password: str) -> dict:
    """
    Analyze password using zxcvbn if available, otherwise fallback heuristics.
    """
    result = {"password": password, "entropy": None, "score": None, "feedback": []}
    if HAVE_ZXCVBN:
        try:
            zx = zxcvbn(password)
            result["entropy"] = zx.get("entropy")
            score = zx.get("score", 0)
            labels = ["Very weak", "Weak", "Fair", "Strong", "Very strong"]
            result["score"] = labels[min(max(int(score), 0), 4)]
            fb = zx.get("feedback", {})
            if fb.get("warning"):
                result["feedback"].append(fb["warning"])
            result["feedback"].extend(fb.get("suggestions", []))
            return result
        except Exception:
            pass

    e = estimate_entropy(password)
    result["entropy"] = round(e, 2)
    result["score"] = entropy_score_label(e)
    if len(password) < 8:
        result["feedback"].append("Password is short — use 12+ characters for better security.")
    if password.lower() in COMMON_WORDS:
        result["feedback"].append("Password is a common word — avoid dictionary words.")
    if re.fullmatch(r"(.)\1+$", password):
        result["feedback"].append("Password is the same character repeated.")
    if re.search(r"(19|20)\d\d", password):
        result["feedback"].append("Password contains a year — years are common and predictable.")
    if not re.search(r"[^\w\s]", password):
        result["feedback"].append("Add symbols to increase strength.")
    return result

# --- Wordlist generation functions ---

def leet_variants(word: str, max_variants: int = 50) -> List[str]:
    variants = set([word])
    letters = list(word)
    positions = [i for i, ch in enumerate(letters) if ch.lower() in LEET_MAP]
    # single and pair substitutions
    for r in range(1, min(3, len(positions)+1)):
        for combo in itertools.combinations(positions, r):
            cand = letters[:]
            for pos in combo:
                ch = letters[pos].lower()
                subs = LEET_MAP.get(ch, [])
                if subs:
                    cand[pos] = subs[0]
            variants.add("".join(cand))
            if len(variants) >= max_variants:
                break
        if len(variants) >= max_variants:
            break
    return list(variants)

def append_years(base: str, years_spec: str = "2000-2025") -> List[str]:
    out = [base, base.lower(), base.capitalize()]
    years = set()
    years_spec = (years_spec or "").strip()
    if not years_spec:
        return out
    if "-" in years_spec:
        try:
            a,b = years_spec.split("-", 1)
            a_i = int(a); b_i = int(b)
            for y in range(a_i, b_i+1):
                years.add(str(y))
        except Exception:
            pass
    else:
        for part in years_spec.split(","):
            p = part.strip()
            if p.isdigit():
                years.add(p)
    for y in sorted(years):
        out.append(base + y)
        out.append(y + base)
        out.append(base + "_" + y)
    return list(dict.fromkeys(out))

def generate_keyboard_patterns(min_len: int = 3, max_len: int = 8, max_entries: int = 2000) -> List[str]:
    """
    Create plausible keyboard walk patterns of different lengths from KEYBOARD_PATTERNS.
    """
    out = set()
    for seq in KEYBOARD_PATTERNS:
        L = len(seq)
        for l in range(min_len, min(max_len, L)+1):
            for i in range(0, L - l + 1):
                piece = seq[i:i+l]
                out.add(piece)
                out.add(piece.capitalize())
                out.add(piece[::-1])
                if len(out) >= max_entries:
                    break
            if len(out) >= max_entries:
                break
        if len(out) >= max_entries:
            break
    return list(out)[:max_entries]

def combine_inputs(inputs: dict, max_entries: int = 20000, years_spec: str = "2000-2025", include_keyboard: bool = False):
    """
    Combine input tokens into candidate words with bounding.
    """
    tokens = []
    for k,v in inputs.items():
        if not v:
            continue
        if isinstance(v, str):
            parts = re.split(r"[,\s]+", v.strip())
            tokens.extend([p for p in parts if p])
        elif isinstance(v, (list, tuple)):
            tokens.extend([str(x) for x in v if x])
    tokens.extend(COMMON_WORDS)
    tokens = list(dict.fromkeys([t for t in tokens if t]))

    expanded = []
    for t in tokens:
        if len(expanded) >= max_entries:
            break
        variants = set()
        variants.add(t)
        variants.add(t.lower())
        variants.add(t.capitalize())
        for v in leet_variants(t, max_variants=12):
            variants.add(v)
        for pre in COMMON_PREFIXES:
            variants.add(pre + t)
            variants.add((pre + t).capitalize())
        for suf in COMMON_SUFFIXES:
            variants.add(t + suf)
        expanded.extend(list(variants))

    # add keyboard patterns if requested
    if include_keyboard:
        kb = generate_keyboard_patterns()
        expanded.extend(kb)

    # limit expanded to avoid explosion
    expanded = list(dict.fromkeys(expanded))[:5000]

    combos = []
    max_combo_len = 3
    try:
        for r in range(1, max_combo_len+1):
            for tup in itertools.permutations(expanded, r):
                cand = "".join(tup)
                combos.append(cand)
                if len(combos) >= max_entries:
                    break
            if len(combos) >= max_entries:
                break
    except MemoryError:
        pass

    # post-process attach small suffixes and years
    final = []
    years = []
    ys = (years_spec or "").strip()
    if ys:
        if "-" in ys:
            try:
                a,b = ys.split("-",1)
                a_i=int(a); b_i=int(b)
                years = [str(y) for y in range(a_i, b_i+1)]
            except Exception:
                years = []
        else:
            years = [p.strip() for p in ys.split(",") if p.strip().isdigit()]

    for c in combos:
        if len(final) >= max_entries:
            break
        final.append(c)
        for suf in COMMON_SUFFIXES:
            if len(final) >= max_entries:
                break
            final.append(c + suf)
        for y in years[:8]:
            if len(final) >= max_entries:
                break
            final.append(c + y)

    # de-duplicate & cutoff
    seen = set()
    out = []
    for w in final:
        if w not in seen:
            seen.add(w)
            out.append(w)
        if len(out) >= max_entries:
            break
    return out

def write_wordlist(words: List[str], outfile: str) -> str:
    Path(os.path.dirname(outfile) or ".").mkdir(parents=True, exist_ok=True)
    with open(outfile, "w", encoding="utf-8", errors="ignore") as f:
        for w in words:
            f.write(f"{w}\n")
    return outfile

# --- Hashcat-rule generation ---

def generate_basic_hashcat_rules(outfile: str, max_rules: int = 200):
    """
    Create a compact set of rules useful for mangling.
    This won't replicate all of hashcat's rules but provides common transforms:
      - append digits/symbols, prepend, toggle case, reverse, duplicate
    """
    rules = []
    # single-char append/prepend (digits + symbols)
    postfix = "0123456789!@#$_-"
    for c in postfix:
        rules.append(f"${c}")  # append
        rules.append(f"^{c}")  # prepend
    # toggle-case / capitalize / reverse / duplicate
    rules.extend(["u", "l", "c", "r", "d"])  # u=uppercase, l=lowercase, c=capitalize, r=reverse, d=duplicate
    # common leet substitutions (use hashcat's notation if needed, but here produce sample simple rules as comments)
    # produce an actual .rule file with comments for clarity
    with open(outfile, "w", encoding="utf-8", errors="ignore") as f:
        f.write("# Basic sample rules generated by pwtool.py\n")
        f.write("# Note: these are simple examples for use with hashcat's --rule-files\n")
        for line in rules[:max_rules]:
            f.write(line + "\n")
    return outfile

# --- Zip bundle creation ---
def create_bundle(zip_path: str, script_path: str, sample_wordlist: str = None, sample_rules: str = None):
    with zipfile.ZipFile(zip_path, "w", compression=zipfile.ZIP_DEFLATED) as z:
        # add script (self)
        z.write(script_path, arcname=os.path.basename(script_path))
        if sample_wordlist and os.path.exists(sample_wordlist):
            z.write(sample_wordlist, arcname=os.path.basename(sample_wordlist))
        if sample_rules and os.path.exists(sample_rules):
            z.write(sample_rules, arcname=os.path.basename(sample_rules))
    return zip_path

# --- CLI / Interactive / GUI glue ---

def print_header():
    print("="*60)
    print("pwtool.py — Password Analyzer & Custom Wordlist Generator".center(60))
    print("defensive testing / education only".center(60))
    print("="*60)

def interactive_menu():
    print_header()
    print("Interactive mode — type the number to choose an action, or 'help' for guidance.")
    menu = [
        "Analyze a password",
        "Generate a custom wordlist (with options)",
        "Generate a basic hashcat .rule file",
        "Create a zip bundle (script + samples)",
        "Start simple GUI (if available)",
        "Quit"
    ]
    for i, item in enumerate(menu, 1):
        print(f"{i}. {item}")
    while True:
        choice = input("\nChoice> ").strip().lower()
        if choice in ("q", "6", "quit", "exit"):
            print("Goodbye.")
            return
        if choice in ("help", "?"):
            print("\nTips:")
            print(" - For analysis enter: 1 then provide password.")
            print(" - For generate enter: 2 then follow prompts (names, pets, years).")
            print(" - Use numeric choices or words like 'analyze', 'generate'.")
            continue
        if choice in ("1", "analyze", "a"):
            pwd = input("Enter password to analyze: ")
            res = analyze_password(pwd)
            print(f"\nPassword: {res['password']}")
            print(f"Entropy: {res['entropy']}")
            print(f"Score: {res['score']}")
            if res["feedback"]:
                print("Feedback:")
                for fb in res["feedback"]:
                    print(" -", fb)
            continue
        if choice in ("2", "generate", "g"):
            name = input("Name(s) (space or comma separated) [leave blank if none]: ")
            pet = input("Pet name(s) (space or comma separated) [leave blank if none]: ")
            birth = input("Birth year(s) or range (e.g. 1990 or 1988,1990 or 1980-1995) [leave blank]: ")
            extra = input("Extra words (comma separated) [optional]: ")
            years = input("Years range to append (e.g. 2000-2025) [default 2000-2025]: ") or "2000-2025"
            max_entries = input("Max number of entries (default 5000): ").strip()
            try:
                max_entries = int(max_entries) if max_entries else 5000
            except Exception:
                max_entries = 5000
            use_kb = input("Include keyboard patterns? (y/N): ").strip().lower().startswith("y")
            out = input("Output filename [default: custom_wordlist.txt]: ").strip() or "custom_wordlist.txt"
            inputs = {"name": name, "pet": pet, "birth": birth, "extra": extra}
            print("\nGenerating — this may take a few seconds depending on limits...")
            words = combine_inputs(inputs, max_entries=max_entries, years_spec=years, include_keyboard=use_kb)
            write_wordlist(words, out)
            print(f"Wrote {len(words)} words to {out}")
            continue
        if choice in ("3", "rule", "rules"):
            rulefile = input("Rule filename to create [default: sample.rule]: ").strip() or "sample.rule"
            generate_basic_hashcat_rules(rulefile)
            print(f"Rule file written to {rulefile}")
            continue
        if choice in ("4", "zip", "bundle"):
            zipname = input("Zip filename [default: pwtool_bundle.zip]: ").strip() or "pwtool_bundle.zip"
            add_sample = input("Include sample wordlist & rules? (Y/n): ").strip().lower()
            include_sample = not add_sample.startswith("n")
            sample_w = "sample_wordlist.txt"
            sample_r = "sample.rule"
            if include_sample:
                # create small sample
                small = ["alice", "bob", "password123", "qwerty!"]
                write_wordlist(small, sample_w)
                generate_basic_hashcat_rules(sample_r)
            create_bundle(zipname, __file__, sample_wordlist=sample_w if include_sample else None,
                          sample_rules=sample_r if include_sample else None)
            print(f"Created bundle {zipname}")
            continue
        if choice in ("5", "gui"):
            try:
                gui_main()
            except Exception as e:
                print("Cannot start GUI:", e)
            continue
        print("Unknown choice — type 'help' for instructions or 6 to quit.")

def cli_main():
    long_desc = (
        "pwtool — Password analyzer and wordlist generator.\n"
        "Examples:\n"
        "  python pwtool.py --analyze \"P@ssw0rd2021!\"\n"
        "  python pwtool.py --generate --name \"Alice Bob\" --pet \"Rex\" --birth \"1990\" "
        "--years 2005-2024 -o mylist.txt --max 10000 --keyboard --rules sample.rule\n"
        "  python pwtool.py --interactive\n"
    )
    parser = argparse.ArgumentParser(
        description="Password Strength Analyzer & Custom Wordlist Generator (pwtool)",
        epilog=long_desc,
        formatter_class=argparse.RawDescriptionHelpFormatter
    )
    group = parser.add_mutually_exclusive_group(required=False)
    group.add_argument("--analyze", "-a", help="Analyze a password string", metavar="PASSWORD")
    group.add_argument("--generate", "-g", help="Generate wordlist (use with other flags)", action="store_true")
    parser.add_argument("--name", help="Name(s) (comma or space separated)")
    parser.add_argument("--pet", help="Pet name(s)")
    parser.add_argument("--birth", help="Birth date or year(s) e.g. 1996 or 1990,1992 or 1980-1995")
    parser.add_argument("--extra", help="Extra words (comma separated)")
    parser.add_argument("--years", help="Years range to append, e.g. 2000-2025 or 2018,2019", default="2000-2025")
    parser.add_argument("--max", type=int, help="Max number of entries for generated wordlist (default 5000)", default=500
    parser.add_argument("-o", "--outfile", help="Output file for wordlist (default custom_wordlist.txt)", default="custom_
    parser.add_argument("--gui", action="store_true", help="Show simple Tkinter GUI")
    parser.add_argument("--interactive", action="store_true", help="Start interactive guided mode")
    parser.add_argument("--keyboard", action="store_true", help="Include keyboard patterns in generation")
    parser.add_argument("--rules", help="Write a simple hashcat-style rule file to the given filename")
    parser.add_argument("--zip", help="Create a zip bundle containing this script and optional samples (provide zip name)"
    parser.add_argument("--include-sample", action="store_true", help="When creating zip, include a tiny sample wordlist a
    args = parser.parse_args()

    if args.interactive:
        interactive_menu()
        return

    if args.gui:
        try:
            gui_main()
        except Exception as e:
            print("GUI cannot be started:", e)
        return

    if args.analyze:
        res = analyze_password(args.analyze)
        print_header()
        print("Password analysis:")
        print(f"  Password: {res['password']}")
        print(f"  Entropy: {res['entropy']}")
        print(f"  Score: {res['score']}")
        if res["feedback"]:
            print("  Feedback:")
            for fb in res["feedback"]:
                print(f"   - {fb}")
        return

    if args.generate:
        inputs = {"name": args.name or "", "pet": args.pet or "", "birth": args.birth or "", "extra": args.extra or ""}
        print_header()
        print("Inputs:", {k: (v or "") for k,v in inputs.items()})
        print(f"Generating up to {args.max} entries (include keyboard patterns: {args.keyboard})...")
        words = combine_inputs(inputs, max_entries=args.max, years_spec=args.years, include_keyboard=args.keyboard)
        print(f"  Generated {len(words)} words (limited to max={args.max}).")
        out = write_wordlist(words, args.outfile)
        print(f"  Wordlist written to: {out}")
        # optionally create rules file if requested in same run
        if args.rules:
            generate_basic_hashcat_rules(args.rules)
            print(f"  Rule file written to: {args.rules}")
        # optionally create zip
        if args.zip:
            sample_w = None
            sample_r = None
            if args.include_sample:
                sample_w = "sample_wordlist.txt"
                sample_r = args.rules or "sample.rule"
                small = ["alice", "bob", "password123", "qwerty!"]
                write_wordlist(small, sample_w)
                generate_basic_hashcat_rules(sample_r)
            create_bundle(args.zip, __file__, sample_wordlist=sample_w, sample_rules=sample_r)
            print(f"  Bundle created: {args.zip}")
        return

    # if only rules requested
    if args.rules:
        generate_basic_hashcat_rules(args.rules)
        print(f"Rule file written to: {args.rules}")
        return

    # if zip requested without generate
    if args.zip:
        sample_w = None
        sample_r = None
        if args.include_sample:
            sample_w = "sample_wordlist.txt"
            sample_r = "sample.rule"
            small = ["alice", "bob", "password123", "qwerty!"]
            write_wordlist(small, sample_w)
            generate_basic_hashcat_rules(sample_r)
        create_bundle(args.zip, __file__, sample_wordlist=sample_w, sample_rules=sample_r)
        print(f"Bundle created: {args.zip}")
        return

    # default: print help with friendly guidance
    parser.print_help()
    print("\nTip: run with --interactive for a guided, conversational flow.")
    print("Examples:")
    print("  python pwtool.py --analyze 'P@ssw0rd2021!'")
    print("  python pwtool.py --generate --name 'Alice' --pet 'rex' --years 2000-2024 --keyboard --rules myrules.rule -o m

# --- Minimal GUI (unchanged behavior, included for convenience) ---
def gui_main():
    try:
        import tkinter as tk
        from tkinter import filedialog, messagebox
    except Exception:
        raise RuntimeError("tkinter is not available in this environment.")
    root = tk.Tk()
    root.title("Password Analyzer & Wordlist Generator (pwtool)")
    tk.Label(root, text="Password to analyze:").grid(row=0, column=0, sticky="w")
    pwd_entry = tk.Entry(root, width=40, show="*")
    pwd_entry.grid(row=0, column=1, columnspan=2)
    result_text = tk.Text(root, height=8, width=70)
    result_text.grid(row=1, column=0, columnspan=4, padx=4, pady=4)
    tk.Label(root, text="Name(s):").grid(row=2, column=0, sticky="w")
    name_entry = tk.Entry(root, width=40)
    name_entry.grid(row=2, column=1, columnspan=2)
    tk.Label(root, text="Pet(s):").grid(row=3, column=0, sticky="w")
    pet_entry = tk.Entry(root, width=40)
    pet_entry.grid(row=3, column=1, columnspan=2)
    tk.Label(root, text="Birth/years:").grid(row=4, column=0, sticky="w")
    birth_entry = tk.Entry(root, width=40)
    birth_entry.grid(row=4, column=1, columnspan=2)
    tk.Label(root, text="Extra words:").grid(row=5, column=0, sticky="w")
    extra_entry = tk.Entry(root, width=40)
    extra_entry.grid(row=5, column=1, columnspan=2)
    tk.Label(root, text="Years range:").grid(row=6, column=0, sticky="w")
    years_entry = tk.Entry(root, width=40)
    years_entry.insert(0, "2000-2025")
    years_entry.grid(row=6, column=1, columnspan=2)
    kb_var = tk.IntVar()
    tk.Checkbutton(root, text="Include keyboard patterns", variable=kb_var).grid(row=7, column=1, sticky="w")
    def do_analysis():
        pwd = pwd_entry.get()
        if not pwd:
            messagebox.showinfo("Info", "Enter a password to analyze")
            return
        res = analyze_password(pwd)
        result_text.delete("1.0", tk.END)
        result_text.insert(tk.END, f"Password: {res['password']}\nEntropy: {res['entropy']}\nScore: {res['score']}\n")
        if res["feedback"]:
            result_text.insert(tk.END, "Feedback:\n")
            for fb in res["feedback"]:
                result_text.insert(tk.END, f" - {fb}\n")
    def do_generate():
        inputs = {"name": name_entry.get(), "pet": pet_entry.get(), "birth": birth_entry.get(), "extra": extra_entry.get()
        years_spec = years_entry.get()
        max_entries = 5000
        words = combine_inputs(inputs, max_entries=max_entries, years_spec=years_spec, include_keyboard=bool(kb_var.get())
        outfile = filedialog.asksaveasfilename(defaultextension=".txt", filetypes=[("Text files","*.txt")], initialfile="c
        if not outfile:
            return
        write_wordlist(words, outfile)
        messagebox.showinfo("Done", f"Wrote {len(words)} entries to {outfile}")
    tk.Button(root, text="Analyze Password", command=do_analysis).grid(row=0, column=3, padx=4)
    tk.Button(root, text="Generate Wordlist", command=do_generate).grid(row=8, column=1, pady=6)
    tk.Button(root, text="Quit", command=root.destroy).grid(row=8, column=2)
    root.mainloop()

if __name__ == "__main__":
    cli_main()
```

Conclusion

The Password Strength Analyzer and Custom Wordlist Generator provides a comprehensive yet educational look into how passwords are evaluated, generated, and tested in cybersecurity. This project encourages better password practices and offers insights into common vulnerabilities such as weak or predictable passwords. It supports future integration with additional tools such as breach databases and advanced rule generators.

Acknowledgment

This project was developed as part of a cybersecurity learning initiative. Special thanks to the open-source community and educational mentors for their contributions to digital security awareness.

■■ Ethical Use Notice

This tool is for **educational and defensive cybersecurity** purposes only. Do **not** use it for unauthorized penetration testing or malicious activities. Always obtain explicit written permission before conducting any form of security testing.

Prepared by: Lekha Sri .G ■
Date: 27 October 2025
Organization: Elevate Lab Internship Project

— End of Document —