Technical Report: Secure Password Generator

Name: Krish Jakher

Registration no. : 25BCE11058

Date: November 23, 2025

Language: Python

Modules Used: secrets, string

1. Executive Summary

The provided script is a command-line utility designed to generate cryptographically strong passwords. Unlike standard random number generators, this script utilizes Python's `secrets` module, making it suitable for security-sensitive applications. The algorithm ensures complexity by enforcing the inclusion of uppercase, lowercase, digits, and special symbols.

2. Code Logic Analysis

The code operates in four distinct phases:

Phase 1: User Input and Validation

- **Mechanism:** Uses a `while True` loop to request input.
- **Validation:**
  - It uses a `try...except ValueError` block to ensure the user inputs a number.
  - It uses a conditional check (`if length >= 4`) to ensure there is enough space to include one of each character type.
- **Observation:** This prevents the program from crashing due to invalid input and ensures the password meets minimum complexity requirements.

Phase 2: Defining the Character Pool

- The script imports standard string constants from the `string` module:
  - `ascii_lowercase` (a-z)
  - `ascii_uppercase` (A-Z)
  - `digits` (0-9)

- `punctuation` (!"#$%&...)
- **Observation:** Using the standard library ensures that no characters are accidentally omitted from the definition.

Phase 3: Password Construction (The Algorithm)

This is the core logic of the generator:

1. **Mandatory Inclusion:** The script explicitly selects one character from each of the four categories (Lower, Upper, Digit, Symbol) and adds them to a list. This guarantees that the password meets standard complexity criteria (e.g., "Must contain at least one symbol").
2. **Filling the Remainder:** The script calculates `remaining_length` and uses a loop to fill the rest of the password slots with random characters from the combined pool.
3. **Shuffling:** Because the first four characters were added in a predictable order (Lower $\to$ Upper $\to$ Digit $\to$ Symbol), the script uses `secrets.SystemRandom().shuffle(password_list)` to mix the characters thoroughly.

Phase 4: Output

- The list is joined into a string using `"".join()`.
- The final result is printed to the console.

---

3. Security Audit: Why is this "Secure"?

The code specifically qualifies as a "Secure" generator due to two main factors:

A. Use of the `secrets` Module

- The code avoids the standard `random` module. The standard `random` module is pseudo-random and deterministic (based on a seed), meaning a skilled attacker could predict the output if they know the state.
- `secrets` generates random numbers using the operating system's entropy source (CSPRNG - Cryptographically Secure Pseudo-Random Number Generator), making prediction practically impossible.

B. Enforced Entropy

- By forcing the inclusion of symbols and numbers, the "search space" for a brute-force attack is significantly expanded compared to a password made only of letters.

---

4. Potential Improvements & Recommendations

While the code is functional and secure, the following improvements would make it more robust for a production environment:

Improvement Area	Suggestion
Reusability	Currently, the function prints the result. It is better practice to `return` the password string so other parts of a program can use it.
Ambiguity	Some fonts make `1`, `l`, and `I` look identical, or `0` and `O`. You might consider adding a filter to remove ambiguous characters for better user experience.
Parameterization	Instead of asking for `input()` *inside* the function, pass the length as an argument (e.g., `def generate_password(length):`). This allows the function to be automated.

---

5. Conclusion

The provided code is a correctly implemented, secure password generator. It successfully handles input errors, adheres to security best practices by using the `secrets` module, and guarantees password complexity through logic that enforces character variety.
