PASSWORD STRENGTH TESTER

By: Aymen MSADDAK, Mouna SAADAOUI, Houssine KHLIF, Aya ARFAOUI

IT 360 - Information Assurance and Security

SECUREPASS

Importance of Password Strength

Consequences of weak passwords:

- Unauthorized access
- identity theft
- financial loss
- privacy breach
- etc..

Why it matters?

- 80% of data breaches involve weak/reused passwords (cite OWASP).
- Entropy measures unpredictability; higher entropy = stronger passwords.

Importance of Password Strength

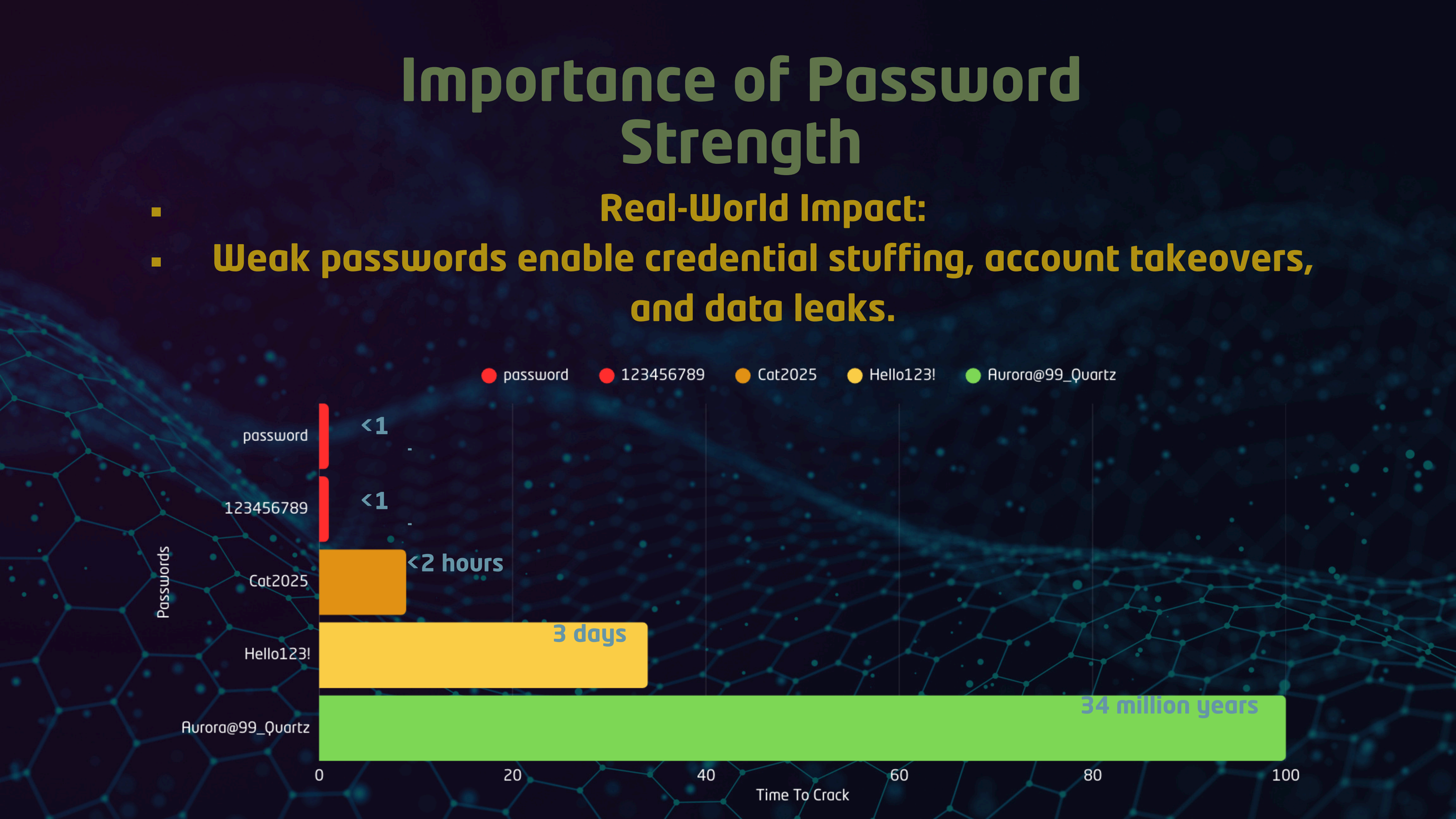
- The Threat Landscape:
  - Weak passwords are the number one cause of data breaches (OWASP).
  - Example: 123456 takes less than 1 second to crack; Thrive#42_Canyon would take centuries.

What is Entropy?

- Entropy Measures unpredictability using log2(R^L).
- Example Calculation:
- Password: Cat2025 → Pool: 26 (lower) + 26 (upper) + 10 (digits) = 62 → Entropy: log2(62^7) ≈ 42 bits (“Moderate”).

Importance of Password Strength

- **Real-World Impact:**
- **Weak passwords enable credential stuffing, account takeovers, and data leaks.**



System Overview

Frontend Components:

- Input Field: Sanitizes input to prevent injection (e.g., stripping whitespace).
- Strength Meter: Dynamically updates color based on entropy (red → green).

Backend Modules:

- Rule Engine: Checks OWASP rules (e.g., check_rules() in rule_engine.py).
- Entropy Calculator: Quantifies strength (e.g., calculate_entropy()).
- Feedback Generator: Maps violations to actionable tips (e.g., FEEDBACK_MESSAGES in feedback_generator.py).

Secure Storage:

- Uses bcrypt (cost factor=12) to hash passwords before logging metadata.

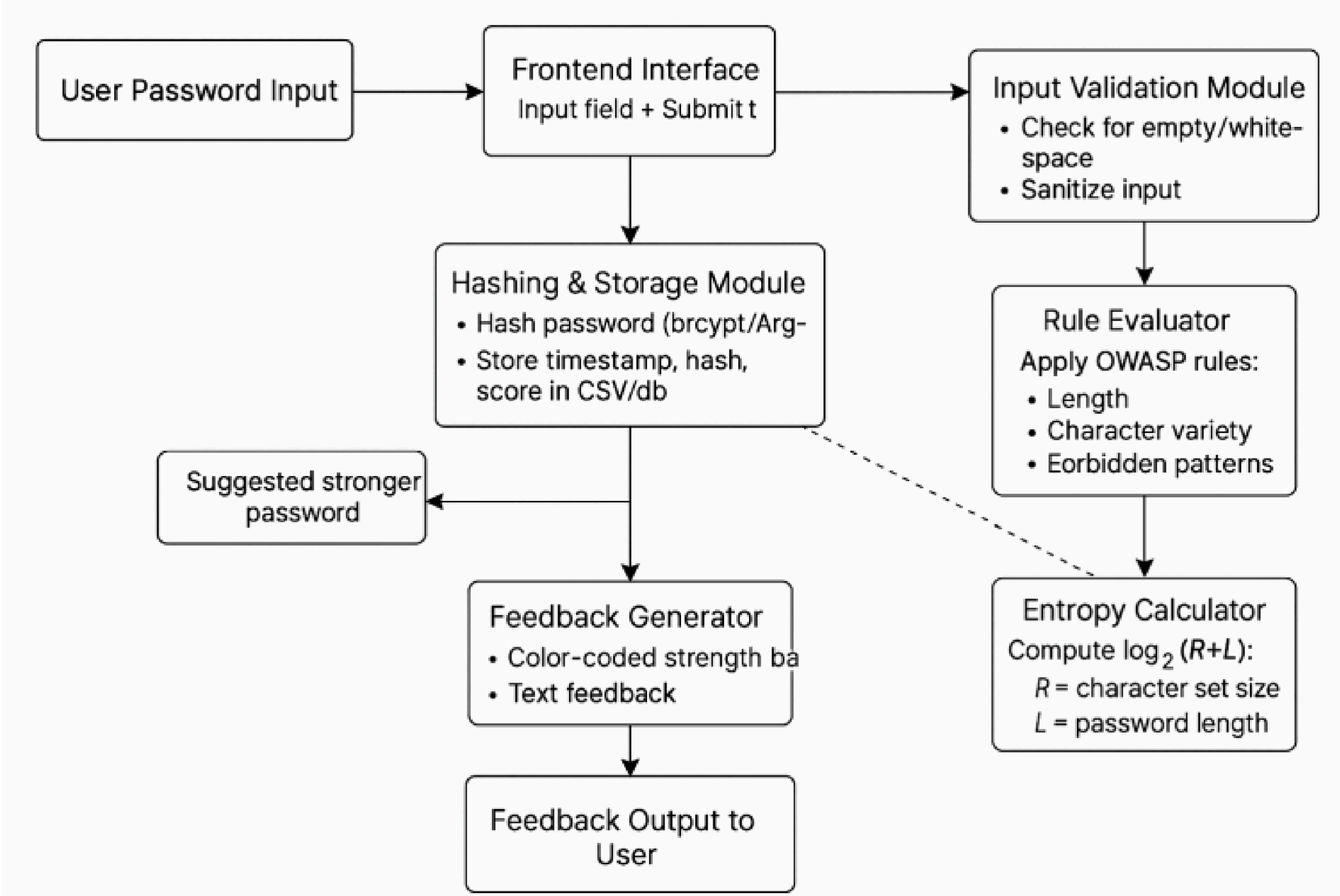


Figure 1: System Design

Step-by-Step Execution Flow

```python
def main():
    password = input("Enter a password to evaluate: ")

    # Step 1: Evaluate the password
    evaluation = evaluate_password(password)
```

1-User Submission:

Input captured via input() in main.py.

2-Validation & Sanitization:

Checks for empty input; trims whitespace.

3-Rule Evaluation:

Code Highlight: check_rules() scans for:
Length (len(password) < 8), character diversity (regex [A-Z], \d), and common patterns (password|1234).

4-Entropy Calculation:

Math Behind It: Pool size (R) = sum of character sets used; length (L) = password length.

Step-by-Step Execution Flow

5-Feedback Generation:

Converts entropy to labels (“Weak”, ”Strong”) and suggests improvements.

```
strength = feedback["strength"]
suggestion_shown = False

if strength in ["Very Weak", "Weak", "Moderate"]:
    suggestion = generate_strong_password()
    print(f"Suggestion: {suggestion}")
    suggestion_shown = True

elif strength in ["Strong", "Very Strong"]:
    print(f"Your password is {strength.lower()}.")
    choice = input("Would you like a strong password suggestion anyway? (yes/no): ").strip().lower()
    if choice == "yes":
        suggestion = generate_strong_password()
        print(f"Suggestion: {suggestion}")
```

6-Secure Logging:

Hashes password with bcrypt, stores entropy, flags, and timestamp in

```
# Add flag to evaluation result for logging
evaluation["suggestion_shown"] = suggestion_shown

# Store results securely in CSV
store_password_result(evaluation)
```

Input Validation & Rule Checks

Key Checks (from

Length:

```python
# 2. Password rule checks
def check_rules(password):
    violations = []

    if len(password) < 8:
        violations.append("too_short")
```

Character Diversity:

```python
if not re.search(r"[a-z]", password):
    violations.append("no_lowercase")

if not re.search(r"[A-Z]", password):
    violations.append("no_uppercase")

if not re.search(r"\d", password):
    violations.append("no_digit")

if not re.search(r"[!@#$%^&*()\-_=+\[\]{};:'\",<.>/?`~]", password):
    violations.append("no_special")
```

Common

```python
if re.search(r"(password|1234|qwerty|azerty|admin)", password.lower()):
    violations.append("common_pattern")
```

Example Input:

Output:

```
Password: pass123
Entropy: 36.19
Flags: ['too_short', 'no_uppercase', 'no_special']
Strength: Moderate
Timestamp: 2025-05-16 00:17:17
```

Entropy Calculation - Behind the Scenes

Formula Breakdown:

$$Entropy = \log_2$$

Strength Categories:

Very Weak (<28 bits), Weak (28-35), Moderate (36-59), Strong (60-127), Very Strong (128+).

Code Implementation (from rule_engine.py):

```python
# 1. Entropy calculation function
def calculate_entropy(password):
    pool = 0
    if any(c.islower() for c in password): pool += 26
    if any(c.isupper() for c in password): pool += 26
    if any(c.isdigit() for c in password): pool += 10
    if any(c in "!@#$%^&*()-_=+[{]};:'\",<.>/?`~" for c in password): pool += 32
    length = len(password)
    return round(math.log2(pool ** length), 2) if pool else 0
```

```python
# 3. Strength category mapping based on entropy
def get_strength_label(entropy):
    if entropy < 28:
        return "Very Weak"
    elif entropy < 36:
        return "Weak"
    elif entropy < 60:
        return "Moderate"
    elif entropy < 128:
        return "Strong"
    else:
        return "Very Strong"
```

Feedback Generation & Suggestions

Feedback

- Violations → Tips: Uses FEEDBACK_MESSAGES dictionary to map flags (e.g., "no_special" → "Add symbols like @ or #").
- Strength → Color: STRENGTH_COLORS maps labels to colors/levels (e.g., "Weak" → orange).

Example:

```
Entropy: 105.49
Flags: ['no_uppercase']
Strength: Strong
Timestamp: 2025-05-16 12:24:04

--- Feedback ---
Strength: Strong
Strength_level: 4
Color: lightgreen
Tips:
- Add uppercase letters (A–Z) to improve strength.
Your password is strong.
```

Code Snippets From feedback_generator.py:

```python
# Feedback message mappings for each rule violation
FEEDBACK_MESSAGES = {
    "too_short": "Your password is too short. Use at least 8 characters.",
    "no_lowercase": "Add lowercase letters (a–z) to improve strength.",
    "no_uppercase": "Add uppercase letters (A–Z) to improve strength.",
    "no_digit": "Include numbers (0–9) to increase complexity.",
    "no_special": "Use special characters (e.g., @, #, $, %, etc.).",
    "common_pattern": "Avoid using common patterns like 'password', '1234', or 'admin'."
}

# Strength color codes and numeric levels
STRENGTH_COLORS = {
    "Very Weak": {"level": 1, "color": "red"},
    "Weak": {"level": 2, "color": "orange"},
    "Moderate": {"level": 3, "color": "yellow"},
    "Strong": {"level": 4, "color": "lightgreen"},
    "Very Strong": {"level": 5, "color": "green"}
}
```

```python
def generate_feedback(evaluation_result):
    flags = evaluation_result.get("flags", [])
    feedback = [FEEDBACK_MESSAGES[f] for f in flags if f in FEEDBACK_MESSAGES]

    strength = evaluation_result["strength"]
    meta = STRENGTH_COLORS.get(strength, {"level": 0, "color": "gray"})

    result = {
        "strength": strength,
        "strength_level": meta["level"],
        "color": meta["color"],
        "feedback": feedback,
        "suggestion": None  # Leave this empty, main.py decides whether to add it
    }

    return result
```

Feedback Generation & Suggestions

Smart Suggestions:

generate_strong_password() combines 3 random words, symbols, and digits.

Code Implementation From feedback_generator.py :

```
l symbols for passphrase-style suggestions

'Ocean", "Maple", "Canyon", "Blink", "Falcon", "Shadow", "Yolk", "Hatch", "
:cho", "Quartz", "Nova", "Summit", "Zephyr", "Crimson", "Meadow", "Flare",
'Breeze", "Starlight", "Granite", "Nimbus", "Comet", "Raven", "Frost", "Inf
.rage", "Vortex", "Harbor", "Blossom", "Pebble", "Gale", "Cinder", "Ivy", "
"Horizon", "Pine", "Quartz", "Serene", "Whisper", "Lyric", "Ember", "Valle

%^&*"
```

```
def generate_strong_password():
    words = random.sample(WORDS, 3)  # pick 3 distinct words
    random.shuffle(words)  # randomize order
    digits = str(random.randint(10, 99))
    symbol = random.choice(SYMBOLS)
    return f"{words[0]}{symbol}{digits}_{words[1]}_{words[2]}"
```

Example:

```
Would you like a strong password suggestion anyway? (yes/no): yes
Suggestion: Pine*41_Whisper_Flare
```

Secure Storage – How We Protect Data

Hashing Process:

Uses bcrypt.gensalt() + hashpw() to create irreversible hashes:

```python
def hash_password(password):
    """Hash the password using bcrypt with automatic salt."""
    salt = bcrypt.gensalt()
    hashed = bcrypt.hashpw(password.encode('utf-8'), salt)
    return hashed.decode('utf-8')
```

CSV Logging:

- Stores only metadata: hashed password, entropy, timestamp, and flags (e.g., "no_uppercase;no_special").
- Security Note: Plaintext passwords are NEVER

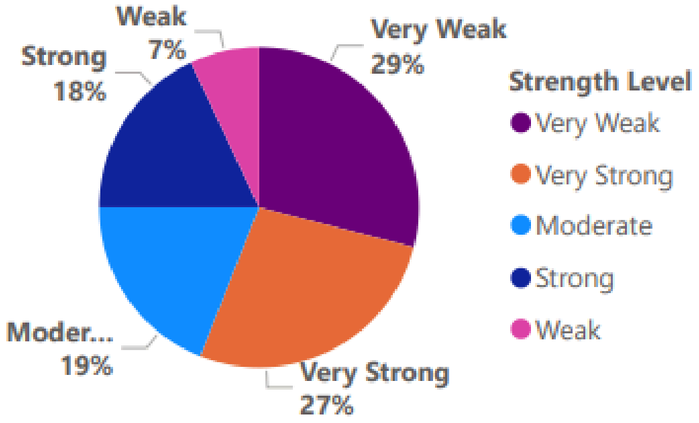
Future-Proofing:

CSV format allows analysis of trends (e.g., common weaknesses).

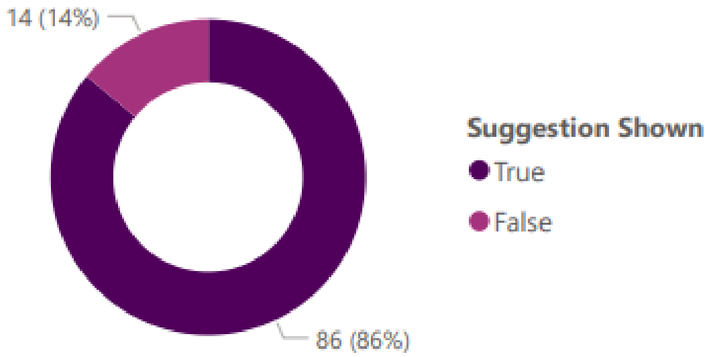
hashed_password	entropy	strength_level	timestamp	flags	suggestion_shown
$2b$12$INhmCZ9c3ykpgnscEg10se8mY	117.98	Strong	5/13/2025 14:18		False
$2b$12$jb9QWp/XxYLUZcYzXQp8k.w8	14.1	Very Weak	5/13/2025 14:20	too_short;no_uppercase;no_digit;no_special	True
$2b$12$bM5MEdsFtHNqjSahGIET3uRN	196.64	Very Strong	5/13/2025 14:21		False
$2b$12$SW24a6eTAJ7NJz4SCcwhcuQb	196.64	Very Strong	5/13/2025 14:22		True

User Password Audit

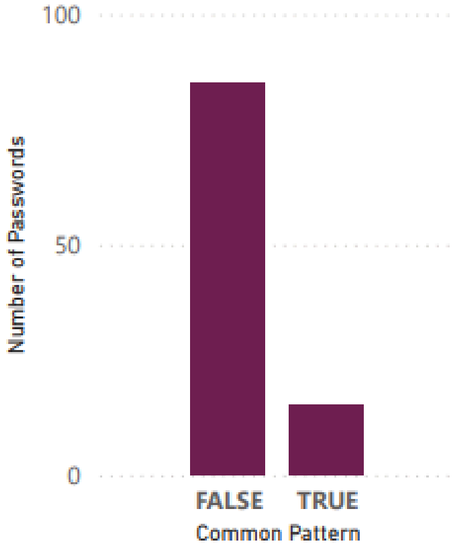
Password Strength Distribution



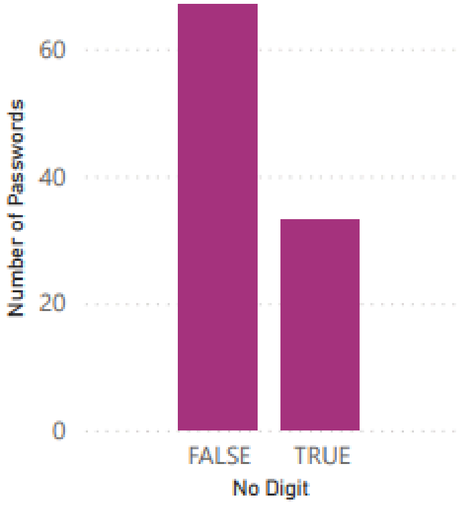
Suggestion Trigger Rate



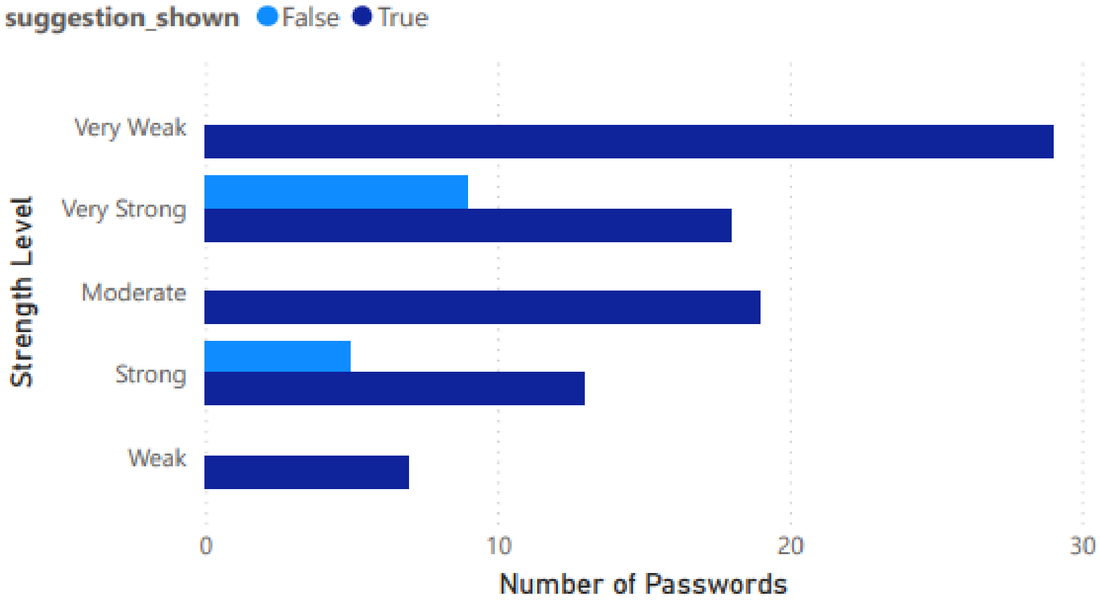
Common Pattern Distribution



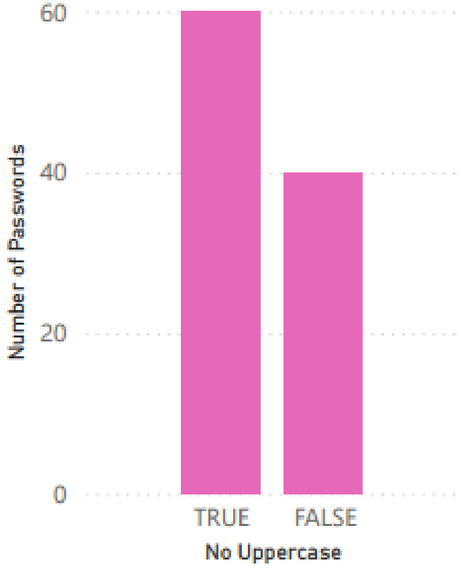
No Digit Distribution



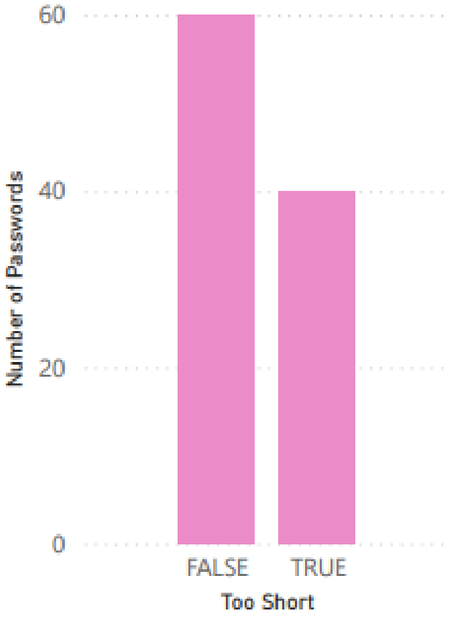
Suggestions by Strength Category



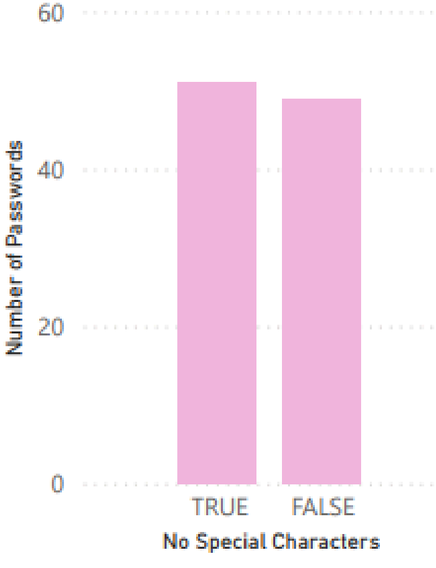
No Uppercase Distribution



Too Short Distribution



No Special Characters Distribution



Password Strength Tester

Enter Password: Test your password...

Evaluate

Strength: None

Feedback:

Future Enhancements:

User Education:

Interactive tutorials on creating strong passphrases.

Machine

Train models to predict password strength based on historical data.

Customizable Password Generation:

Add more features to incorporate user preferences in generated passwords.

THANK YOU!