1. **Double Length PRF**

   The algorithm is as follows. $Setup'$ outputs a key $K_2 \in \{0,1\}^n$. $Eval'(K_2, x) = y_1 \mid y_2$ where $y_1 = Eval(K_1, x)$ and $y_2 = Eval'(K_2, x)$ from the challenger. The output is by construction $2l$ long.

   We then prove that $Setup'$ and $Eval'$ is secure. Suppose there exists an algorithm A that returns the correct b' of whether the input string of length $2l$ is random or pseudo-random. The two reduction schemes are shown in Figure 1 and 2.

   For the first reduction, we construct a new attacker with Algorithm A who passes through x of length l (Figure 1). Algorithm B in the middle passes it along to the PRF challenger. With K_1 from $Setup$, it returns y_1, which is then combined with y_2 and sent to the attacker. After that, Algorithm B passes b' from the attacker to the challenger.

   Since $Setup$ and $Eval$ is a PRF, $Adv_A = negl(n)$. So now we have $Adv_B = Pr[B \to 1 \mid b' = 1] - Pr[B \to 1 \mid b' = 0] = Pr[A \to 1 \mid b' = 1] - Pr[A \to 1 \mid b' = 0] = Adv_A = negl(n)$. We denote a unit of l bits as R if they are random and E if they are pseudo-random. $y_1 \mid y_2$ is either RE or EE, so the attacker does not have non-negligible advantage in distinguishing RE from EE, $Pr[A \to 1 \mid RE] - Pr[A \to 1 \mid EE] = negl(n)$.

   For out second reduction, most part is identical as the first one except that B' passes $y_2 \mid y_1$ where y_2 is random bits (Figure 2). Similarly, $Adv_{B'} = negl(n)$ and $y_2 \mid y_1$ is either RR or RE. The attacker does not have non-negligible advantage in distinguishing RR from RE. $Pr[A \to 1 \mid RR] - Pr[A \to 1 \mid RE] = negl(n)$.

   If we sum up the two equations, $Pr[A \to 1 \mid RE] - Pr[A \to 1 \mid EE] + Pr[A \to 1 \mid RR] - Pr[A \to 1 \mid RE] = Pr[A \to 1 \mid RR] - Pr[A \to 1 \mid EE] = negl(n)$, which means the attacker is not able to distinguish between RR and EE. Thus, $Setup'$ and $Eval'$ is a PRF.

2. **Discrete Logarithm**

   No, the discrete logarithm problem is not hard anymore with the function $O(g^a, g^b)$. In a logarithm problem, we are supposed to find a when given $g^a = h$ where g and h are elements in G. The function $O(g^a, g^b)$ acts as a compactor of a and b, with which we can perform a binary search over the interval $0 \leq a \leq p-1$.

   We first find the midpoint of the interval, $\lfloor p/2 \rfloor$, and check $O(g^a, g^{\lfloor p/2 \rfloor})$. If it outputs 0, we know that $a < \lfloor p/2 \rfloor$ and will keep searching the first half recursively. Otherwise, if the other half contains more than one element, we keep searching; else, we find a to be that remaining element a' in the interval. The algorithm is guaranteed to terminate because and that the size of the search space keeps decreasing until 1. The result is correct at its termination because a' is the smallest one of all elements $\leq a$ and that $0 \leq a \leq p-1$.

   As each comparison cuts the search space into half until we find $a' = a$, the time complexity of the search is $O(logp)$ times the complexity of the O function, $O(log^2 p)$.

3. **Two-Query PRP**

   My construction of PRP returns $f(x) = ax + b \bmod q$ where $a, b \in Z_p^*$ and $a \neq 0$.

   $f(x)$ is a permutation, proved as follows. Suppose $xa + b = x'a + b$, we obtain $a(x - x') \equiv 0 \bmod p$. Since $a \neq 0$, we get $x - x' \equiv 0$, which can be rewritten as $x \equiv x' \bmod p$. Thus, $f(x)$ is unique for x and $f(x)$ is a permutation.

   To prove that $f(x)$ belongs to PRP, we need to show $Pr[f_k(x_1) = c_1, f_k(x_2) = c_2]$ is true for all pairs of queries x_1 and x_2. By construction, we have $c_1 = ax_1 + b$ and $c_2 = ax_2 + b$. The case when $x_1 = x_2$ is trivially true, so we assume $x_1 \neq x_2$. When c_1, c_2, x_1, and x_2 are all known, we can solve for a and b.

$$a' = \frac{c_2 - c_1}{x_2 - x_1}$$

$$b' = c_1 - x_1 * \frac{c_2 - c_1}{x_2 - x_1}$$

For every pair of (x_1, x_2), the pair (a, b) is unique. Since a and b are sampled randomly,

$$Pr[f_k(x_1)] = c_1, f_k(x_2) = c_2] = Pr[a = a', b = b'] = \frac{1}{p(p-1)}$$

for every pair of (a, b). Thus, the design is unconditionally true.

4. **Strong Password**

I would consider the following threats when designing a program that prompts users to generate strong passwords: users who attempt to enter easily-breakable passwords and attackers who try to break users' passwords. Considering both scenarios, I would generate a series of rules that define a strong password and then force users to meet these rules when creating passwords.

My assumptions are as below. On the one hand, users tend to enter passwords that are easy to remember, which are usually the ones easy to break. On the other side, we assume that attackers could only brute force the password or just guess users' passwords. Therefore, the main idea of all rules is to push users to create more complex passwords, or the ones with higher entropy. For practical purpose though, passwords should have a limited length of about 10-14 characters. Since we now have an upper-bound of how chaotic a password string could be, we define password entropy to be the number of possibilities under a certain set of rules.

My design assumes a certain alphabet size for each character in the password. It would allow English letters, numbers, and some commonly seen special characters. To further maximize variations, passwords are case sensitive so that an uppercase "A" is different than a lowercase "a. " If an alphabet contains 20 special characters and a password has a length, each character has $26 + 26 + 10 + 20 = 82$ possibilities. Based on the assumptions talked above, I would validate user inputs according to following rules that maximize the password entropy.

First, a strong password must meet some minimum required length. If the alphabet size is s and the password must be at least l, then there are s^l possibilities. To prevent attackers from brute force search, we prefer longer passwords. Users tend to enter short passwords at their first several tries, so some hints should display to remind them to increase the length of their passwords. The length, however, should not be too long or users would not be able to remember their passwords. I would set the minimum to a reasonable number, like 8.

Second, a strong password should adopt a combination of letters, numbers, and special characters. For example, we could set up a rule stating that a password must contain at least one letter, one number, and one special character. This is to ensure that users make good use of the size of the alphabet. If a user only use numbers in the password, the password entropy is only 10^l as if the alphabet only contains 10 characters. The rule would boost the variety of user input and increase the difficulty for brute-force attackers.

Third, my design would prevent users from including some common strings in their passwords, such as "123456," "password," and "qwerty." This rule is to increase the difficulty for attackers who use a common-word dictionary to guess users' passwords. If users put down their passwords with certain structures and units, it decreases possibilities of other combinations of characters and thus becomes more risky. If we could exclude certain common units in passwords, users have to turn to combinations that are not as common, which is beneficial in preventing attackers from guessing the right passwords.

So far we have considered user behaviors and attacker types and come up with corresponding rules to increase the entropy held within limited password lengths. Hope the design has well reduced the threats from a practical perspective.