12.10.3 Solution:

Without losing of generality, we can assume the sequence \( \{ P_i \} \) is non-increasing. We can also assume the attacker knows the probability distributions of the passwords. The strategy of the attacker would be to try the most likely password first. There are TG guesses within the timeframe. So, the probability of successfully guessing a password is

\[
P = \sum_{i=1}^{TG} P_i
\]

To determine \( P_i \) in a real system, a central authority can collect all the passwords people use and generate a histogram of all these passwords. Since it is infeasible to collect all passwords from all people in the world, we can ask a statistician to design a sampling method, so that we can collect only a fraction of all the passwords used. Based on the sampling results and statistical methods, we can estimate \( P_i \) for each string.

We can also use the existing research by Klein [573], to assign \( P_i \) differently according to whether the string is a dictionary word, common name, etc.

Theorem 12-1 tells us that it is most difficult to guess a password when the passwords are uniformly distributed. So, a system should randomize passwords selection as much as possible to ensure maximum security.

(Courtesy of Matthew Henkler)

12.10.11) The main problem that exists if the seeds for S/Key are not chosen randomly is that an attacker could predict what the initial seed is for the next sequence of passwords. For example, if it was known that the initial seed was chosen based off of the current system time, the attacker would have a much smaller keyspace to do a brute-force attack on, which would significantly reduce the time needed to compute the sequence of passwords.

If the seed provided to S/Key is purely random however, the full keyspace of MD4 (5) must be possibly tested in order to break the system, as MD4 and MD5 are considered to be one-way hash functions.

The assumption that the seeds are purely random may be a tad bit hopeful, however. Generating a purely random number from scratch on a system can be a daunting task, as current technology does not provide a method for generating a purely random number, unless expensive equipment is connected, such as a radioactive decay detector. However, current methodologies for generating numbers that for most purposes are random enough to use with cryptographic algorithms, so for all intents and purposes, this assumption will be fairly strong as long as good algorithms for generating good pseudo-random numbers (i.e. through cryptography or a good pseudo-random number generator) are available.
Principle of Least Privilege – This approach violates this principle, as the user is given vastly more privileges than necessary to complete its task when the password file cannot be opened. In this case, the user is given all the rights of the superuser, which is more than enough rights to fix the problem of not being able to open the file. Secondly, su does not specify which rights of the other user to be granted to the requester. As such, even without the above design modification, su violates the principle of least privilege as once the user successfully authenticates through su, they are given all of the rights and privileges associated with the other user.

Principle of Fail-Safe Defaults – This approach violates this principle. This is because the default action that is taken upon not being able to read the password file is to enable superuser access, when instead a better fail-safe default would instead deny access instead of granting extra privilege.

Principle of Economy of Mechanism – This approach violates this principle. This is because the mechanism of this approach interacts with the rights and privileges granted to the superuser of the system. As such, one must make many possibly incorrect assumptions in order to ensure the security of this approach. A simpler approach that does not possibly grant a user superuser rights would be more economical.

Principle of Complete Mediation – Though this approach does not directly violate the principle of complete mediation, the su command itself does. This is because once a user is given the rights of another user through the su command on most UNIX systems, it is not checked on every successive access to ensure that the user is still allowed those rights.

Principle of Open Design – This approach does meet the principle of open design. How the system works is made public to the user of it.

Principle of Separation of Privilege – This approach violates the principle of separation of privilege in two ways. First, if the su command succeeds, the user obtains permissions based solely on a single condition – that of knowing the correct password. Secondly, it violates this principle when the password file is not present, as the user is able to obtain superuser permissions based on that sole condition.

Principle of Least Common Mechanism – This approach does not directly violate this principle.

Principle of Psychological Acceptability – This approach meets this principle. This is because if for some reason the password file is non-existent, a user would not want to be denied access as this would make the system more difficult to access than if the security mechanisms were not present. Thus, giving the user the ability to remedy the problem meets the principle of psychological acceptability.
14.1(b) The server needs to store (username, hash) pairs.
   (c) In this case the cookie is used both as an authentication and to store state.

18.6.7 Solution:
The software developed is not “high assurance”. This is because the extreme
programming development methodology leaves the requirements open until the project is
complete. Leaving the requirement open does not ensure that security requirements be
properly implemented into the system.

If the company did not analyze threats and do not have the appropriate security
requirements in place, the software developed would not be able to implement those
security requirements. I would suggest the management to provide assurance in
requirements, design, implementation, and operation and management.

For example, if this software is used for online shopping, and it functions correctly
according to programmer’s testing. But if it did not consider the customer’s credit card
information should be protected during the requirement phase, the programmer might
have coded the software in such a way that allows the customer’s credit card information
to transmit in clear. This software is not high assurance at all.