Problem 1 (10 pts)

1. (5pts) The following is the attribute for a file in Unix

    -r-srw---x  1 simon   fac   0 Jul 18 21:55 get_score

User “simon” and “jay” belong to group “fac”, user “daniel” belongs to group “student”.

List all privileges (r,w,x) each of the three users have on the file (note: the setuid bit “s” also indicates execution privilege x).

simon: read, execution

jay: read, write

daniel: execution

2. (5pts) When user jay executes the get_score program above, which user(s)’s privilege can the process have? What if user “simon” executes the program? Explain your answers.

   Hint: the program may manipulate the various uid fields in the process’s control block as we learned in the setuid paper.

   Answer: When user jay executes the program, because the setuid bit is set, the process will have euid set to simon’s uid so the process will have simon’s privilege. In addition, the real uid field will be set to jay’s uid, and the program can swap it into the euid, so the process could also have jay’s privilege.

   When simon executes the program, both uid fields will be set to simon’s uid, so it only has the owner, simon’s privilege.
Problem 2 (10 pts)

A program has a buffer overflow vulnerability on the stack. The operating system adopts stack address randomization as we explained in class. It also uses canary to detect overridden saved EIP. For each of the following exploits, explain whether it will work on this system or not; explain your answers.

1. (3 pt) A buffer overflow exploit to overwrite the saved EIP on the stack to point to the shellcode injected on the stack.

   **Answer:** This exploit cannot work. Before overwriting saved EIP on the stack the canary word would be corrupted, therefore the program will be able to detect the attack. Also because of the ASLR protection, the stack address changes every time when the program launches; the attack is unlikely to accurately redirect the program to a reliable address of shellcode.

2. (3 pt) A buffer overflow exploit to corrupt the local variables on the stack frame and creates a situation that allows the attacker to modify any arbitrary memory location with arbitrary value. Then use this to modify the saved EIP on the stack to point to a “jump code” in a fixed memory location to indirectly jump into the buffer where the shellcode is injected.

   **Answer:** This attack will not work. Even though the attackers can modify any arbitrary memory location, the attacker is not able to predict where the saved EIP is on the stack due to stack address randomization.

3. (4 pt) A buffer overflow exploit to corrupt the local variables on the stack frame and create a situation that allows the attacker to modify any arbitrary memory location with arbitrary value. Then use this to modify the entrance address of a system function that will be called by the program, to point to a “jump code” in a fixed memory location to indirectly jump into the buffer where the shellcode is injected.

   **Answer:** This attack can work reliably. Because the attacker can overwrite arbitrary memory, the attacker can modify the entrance address of a system used function no matter how random the stack address is. Because the function
will be called by the program, there is no need to overflow the buffer on the stack to reach the saved EIP, so canary will not protect this.

Problem 3 (6 pts)

Which of the following OS protection mechanisms can be circumvented by return into libc attack. Explain your answers for both the yes and no cases.

1. Memory address randomization for stack.

**Answer:** Yes, this protection can be circumvented by return-into-libc attack. Even the stack memory address is randomized, the attacker can still figure out how long a string is needed to exactly overwrite the saved EIP, find a reliable entrance address of a libc function, which is not randomized. Therefore it is possible for the attacker to take control and hijack the program.

2. Memory address randomization for shared libraries.

**Answer:** No, this protection cannot be circumvented by return-into-libc attack. Return-into-libc attack requires a reliable entrance address for shared libraries. Even if we assume the attacker already has a way to overwrite saved EIP, memory address randomization for shared libraries hinders attacker's ability to find a reliable entrance address in the libc library.

3. Non-executable stack

**Answer:** Yes, this protection can be circumvented by return-into-libc attack. Even though the stack is non-executable, the shared library, which contains necessary function calls such as ‘system()’ has to be executable. By using return-into-libc attack, the attacker jumps to existing executable code, which circumvents the limitation of non-executable stack.

Problem 4 (4 pts)

In earlier days web service programs were often run as root user. Later on those programs are run by default as a user with limited privilege on the system, e.g., it can only access a portion of the file system that contains the website’s data. From a security perspective, explain why this change happened.

**Answer:** Root user is the most powerful user account in a system. When web service programs run as root, if there are vulnerabilities, such as buffer overflow, in the program, attacker can take advantage of such vulnerabilities to alter the normal control flow and execute arbitrary code on behalf of the root user, which basically
gives the attacker all privileges on the system. By running web service programs as a user with limited privilege, even if such attacks happen, the attacker can only get as much privilege as the limited user. The attacker has to find more vulnerabilities in order to take total control of the whole system, which makes his/her job more difficult, hence the system is more secure.

**Problem 5 (10 pts)**

Why are password hashes stored on the system? Explain both the need of storing the hashes, and the security of storing hashes. What properties of hash functions are important for each of the aspects?

**Answer:** System stores password hashes, because the system needs evidence to verify user provided password in order to authenticate user. The reason why storing password hashes rather than plaintexts or encryption is that the hashing process is not reversible. By storing password hashes even the system is compromised it is hard to get the clear-text passwords. This is due to the pre-image resistance property of hash. Hash’s collision resistance property ensures that if the calculated hash matches the stored one, the user input must be identical to the correct password.

**Problem 6 (10 pts)**

In order to ensure data transmitted through Internet is not tampered with, Alice and Bob conduct the following protocol in communication.

1. Alice sends a message \( m \) along with its digest \( c = H(m) \) where \( H \) is a cryptographic hash function.
2. Upon receiving \( m \) and \( c \), Bob verifies that \( c = H(m) \).

Can this protocol protect the message’s integrity during transmission? Explain. If you think this protocol does not achieve the intended objective, can you modify it to make it work?

**Answer:** This protocol cannot protect the message’s integrity during transmission. Cryptographic hash function is not encryption. Those hash algorithms are well known and publicly available. Suppose there is a man in middle capable of intercepting messages between Alice and Bob. The attacker can send arbitrary massage \( m' \) and using the same hash function \( c' = H(m') \) and send \((m', c')\) to Bob. When Bob receives the message, there is no way for him to be sure whether the message is from Alice or not.

In order to guarantee the message’s integrity, Alice and Bob can exchange their public keys offline (or through other secure protocols). Alice and Bob can sign messages using his/her own private key and send the plaintext and the digital signature together
through the network. When the other side receives the message he/she can verify signature on the message using the sender's public key.

Alternatively, since we only care about the message’s integrity, Alice and Bob can pick a shared key $K$ offline (or through other secure protocols). To communication, they can send message $m$ with $h$, where $h = \text{MAC}(m, K)$, when the other side receives the message, he/she can verify the integrity by applying the same MAC calculation with message $m$: $h' = \text{MAC}(m, K)$, if $h = h'$ then the receiver can be assured the integrity of message is not compromised.