알고리즘 적용 기획서

김제영, 정여민

October 7, 2023

Institution: SSAFY 10th

1 해싱을 통한 비밀번호 암호화

1.1 적용 알고리즘

Hash 단방향 해시 함수는 어떤 수학적 연산(또는 알고리즘)에 의해 원본 데이터를 매핑시켜 완전히 다른 암호화된 데이터로 변환시키는 것을 의미한다. 이 변환을 해시라고 하고, 해시에 의해 암호화된 데이터를 다이제스트(digest)라고 한다.

Salt 솔트란 해시함수를 돌리기 전에 원문에 임의의 문자열을 덧붙이는 것을 말한다. 해시함수는 입력값이 약간만 달라져도 전혀 다른 아웃풋을 내기 때문에, 다이제스트를 알아낸다 하더라도 password 를 알아내기 더욱 어려워진다. 그리고 사용자마다 다른 Salt 를 사용한다면 설령 같은 비밀번호더라도 다이제스트의 값은 다르다. 이는 결국 한 명의 패스워드가 유출되더라도 같은 비밀번호를 사용하는 다른 사용자는 비교적 안전하다

Key-Stretching 해시 함수를 여러번 돌려서 다이제스트를 얻어내는 방법으로, 최종 다이제스트를 얻는데 그만큼 시간이 소요되기 때문에 브루트포스 공격에 대비할 수 있다.

1.2 알고리즘 개요

사용자가 로그인을 하기 위해 아이디와 비밀번호를 입력하면 서비스는 DB에서 해당 데이터에 맞는 사용자가 존재하는 지 확인한 후 로그인 성공 여부를 알려준다. 이러한 비밀번호를 저장해 놓은 테이블을 Rainbow Table이라고 하는데, 이 테이블이 해커의 손에 넘어가면 심각한 보안 침해 문제를 초래할 수 있다. 따라서 비밀번호를 암호화하여 저장하는 것이 중요하며, 이때 사용되는 것이 해시 함수이다.

1.2.1 해시

Figure 1과 같이 해시함수는 단뱡향으로 암호화가 가능하지만 복호화는 불가하다. 단방향 해시 함수는 어떤 수학적 연산(또는 알고리즘)에 의해 원본 데이터를 매핑시켜 완전히 다른 암호화된 데이터로 변환시키는 것을 의미한다. 이 변환을

Figure 1: 해시함수를 사용한 비밀번호 암호화

해시라고 하고, 해시에 의해 암호화된 데이터를 다이제스트(digest)라고 한다. 비밀번호를 해시함수에 돌려서 다이제스트를 생성하고 이 다이제스트를 비밀번호 대신 DB에 저장하는 방법을 사용할 수 있다.

1.2.2 한계점

그러나 단방향 해시함수에는 명백히 한계점이 존재한다. 첫째, 동일한 메시지는 동일한 다이제스트를 갖는다. 같은 해시함수에 대해 해커들이 여러 값들을 대입해보면서 얻었던 다이제스트들을 모아놓은 리스트를 레인보우 테이블(Rainbow Table)이라 부른다. 해커가 구한 다이제스트 값이 레인보우 테이블에 있으면 비밀번호가 노출된 것과 다름없다. 둘째, 브루트포스 공격에 취약하다. 해시함수는 원래 빠른 데이터 검색을 위한 목적으로 설계된 것으로 다이제스트를 얻는 과정도 빠르다. 그 말인 즉, 해커 역시 다이제스트를 빠르게 얻을 수 있으므로 무작위로 데이터를 계속 대입하여 얻은 다이제스트와 해킹할 대상의 다이제스트를 계속 비교하여 비밀번호를 알아낼 수 있게 된다.

1.2.3 보완점

단반향 해시함수를 보완하는 방법에는 키 스트레칭(Key-Stretching)과 솔트(Salt)를 들 수 있다.

Key-Stretching 동일한 해시함수를 여러번 돌려 다이제스트를 얻어내는 방법이다. Figure 2와 같이 비밀번호를 입력으로 하여 해시함수를 돌려 얻어낸 다이제스트를 다시 해시함수에 넣어 다이제스트를 만들어내는 방법을 N회 반복하여 최종적으로 얻어낸 다이제스트를 DB에 저장하는 방법이다. 사용자의 경우 패스워드를 입력하고 일치여부를 확인 할 때 0.2 0.5 초만 걸려도 크게 문제가 없다. 그러나 임의의 문자열을 무차별 대입하는 해커 입장에서는 1초에 10억번의 다이제스트를 얻을 수 있었으나 다이제스트를 얻기 까지의 시간을 지연시켜 이제는 한 횟수당 0.2 0.5초가 걸리기 때문에 매우 치명적이다. 즉, 브루트포스를 최대한 무력화하기위한 방법인 것이다.

Salt 솔트란 해시함수를 돌리기 전에 원문에 임의의 문자열을 덧붙이는 것을 말한다. 이렇게 하면 다이제스트를 알아낸다 하더라도 비밀번호를 알아내기 더욱 어려워진다. 그리고 사용자마다 다른 Salt 를 사용한다면 설령 같은 비밀번호더라도 다이제스트의 값은 다르다. 이는 결국 한 명의 패스워드가 유출되더라도 같은 비밀번호를 사용하는 다른 사용자는 비교적 안전하다는 의미기도 하다.

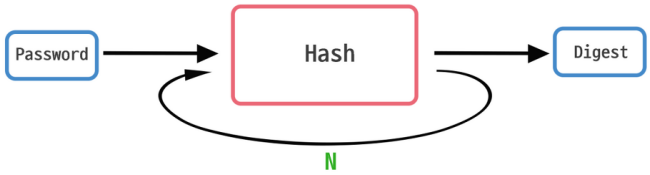

Figure 2: Key-Stretching을 적용한 해시

Figure 3: Salt를 적용한 해시

1.3 적용 서비스 및 개발 개요

사용자가 회원가입을 할 때 입력한 비밀번호에 임의의 문자열 salt를 붙여 해시 함수(SHA-256)을 돌려 얻은 다이제스트를 salt와 함께 DB에 저장한다. 이후 로그인을 할 때, 사용자가 입력한 아이디를 기반으로 DB에서 해당 사용자의 salt와 다이제스트를 가져온다. 사용자가 입력한 비밀번호에 salt를 더해 해시함수를 돌린 것과 DB에서 가져온 다이제스트가 동일할 때 로그인이 성공하며, 다를 경우 로그인이 실패한다. Figure 4는 회원가입을 할 때와 로그인을 할 때 암호화를 적용한 부분이다.

```java
@Override
public int joinMember(MemberDto memberDto) throws Exception {

    // 여기서 패스워드 + salt 를 해시 돌려서 다이제스트와 salt를 DB에 같이 저장
    String Salt = CryptoPW.getCryptoPW().getSALT();
    String HashDigest = CryptoPW.getCryptoPW().Hashing(memberDto.getUserPwd().getBytes(), Salt);

    memberDto.setUserPwd(HashDigest);
    memberDto.setSalt(Salt);


    return memberDao.joinMember(memberDto);
}

@Override
public MemberDto loginMember(String userId, String userPwd) throws Exception {

    // userId로 멤버를 가져와서 userPwd를 해시 돌린 것과 비교
    MemberDto temp = SearchMemberById(userId);


    // 로그인 성공
    if(temp.getUserPwd().equals(CryptoPW.getCryptoPW().Hashing(userPwd.getBytes(), temp.getSalt()))) {
        return memberDao.loginMember(userId, CryptoPW.getCryptoPW().Hashing(userPwd.getBytes(), temp.getSalt()));
    }

    // 로그인 실패
    return null;
}
```

Figure 4: 회원가입을 할 때 비밀번호를 암호화 하는 과정

또한 Figure 5에서 볼 수 있듯 비밀번호를 변경한 경우도 DB에 저장되는 다이제

스트를 업데이트 시켜줘야한다.

```java
@Override
public int modify(String userId, String userPwd) throws Exception {

    MemberDto temp = SearchMemberById(userId);
    userPwd = CryptoPW.getCryptoPW().Hashing(userPwd.getBytes(), temp.getSalt());


    return memberDao.modify(userId, userPwd);
}
```

Figure 5: 비밀번호 수정을 할 때 해시 다이제스트를 업데이트 하는 과정

보안에 사용된 메서드는 크게 salt를 생성하는 것과 hash를 돌리는 것 두 개가 있다. getSALT 메서드는 임의의 문자열을 생성하며 반환한다. Hashing 메서드는 Byte 배열로 입력받은 비밀번호와 salt를 사용하여 해시함수를 돌리는데 key-stretching 을 사용하여 브루트포스 공격에 대한 내성을 강화했다.

```java
// 비밀번호 해싱
public String Hashing (byte[] password, String Salt) throws Exception{
    MessageDigest md = MessageDigest.getInstance("SHA-256");     // sha 256 사용

    // key - stretching
    for(int i=0; i < 10000; i++) {
        String temp = Byte_to_String(password) + Salt; // 패스워드와 salt를 합쳐 새로운 문자열 생성
        md.update(temp.getBytes()); // temp의 문자열을 해싱하여 md에 저장
        password = md.digest(); // md 객체의 digest를 얻어 password 갱신
    }
    return Byte_to_String(password);
}

// SALT 값 생성
public String getSALT() throws Exception {
    SecureRandom rnd = new SecureRandom();
    byte[] temp = new byte[SALT_SIZE];
    rnd.nextBytes(temp);

    return Byte_to_String(temp);
}

// 바이트 값을 16진수로 변경해준다
public String Byte_to_String(byte[] temp) {
    StringBuilder sb = new StringBuilder();
    for (byte a : temp) {
        sb.append(String.format("%02x", a));
    }
    return sb.toString();
}
```

Figure 6: 암호화와 관련된 코드

2 정렬을 이용한 조회수 기반 관광지 추천

2.1 적용 알고리즘

QuickSort 한 개의 피벗을 기준으로 하여 좌측에는 피벗보다 작은 값, 우측에는 피벗보다 큰 값들이 오도록 계속해서 정렬해나가는 알고리즘이다. 평균적으로 $O(NlogN)$의 시간복잡도를 가지고 정렬을 수행한다.

BurbleSort 두 개의 요소를 비교하여 정렬을 수행하는 알고리즘이다. 한 번 수행 할 때 마다 가장 우측에 제일 작은 값(내림차순 정렬 기준)이 위치함을 보장한다. 항상 O(N^2)의 시간복잡도를 가지고 있다.

2.2 알고리즘 개요

2.2.1

QuickSort 불안정 정렬에 속하며, 다른 원소들과의 비교만으로 정렬을 수행하는 비교 정렬이다. 추가적인 메모리 공간이 필요하지 않다. 피벗만 잘 선택한다면 평균적으로 O($NlogN$) 시간에 동작한다. **BurbleSort** 안정 정렬에 속하며, 서로 인접한 두 원소를 검사하여 정렬하는 알고리즘이다. QuickSort와 마찬가지로 추가적인 메모리 공간이 필요하지 않다. 항상 O(N^2)의 시간복잡도를 가진다.

2.2.2 정렬 방법

- **QuickSort**
  a. 리스트 안에 있는 한 요소를 피벗으로 선택한다.
  b. 피벗을 기준으로 피벗보다 작은 요소들은 피벗의 왼쪽으로, 큰 요소들은 피벗으 오른쪽으로 이동시킨다.
  c. 피벗의 왼쪽에 있는 리스트와, 오른쪽에 있는 리스트를 분할 정복을 이용해 2번 과정을 계속해서 수행하며 정렬을 수행한다. (부분 리스트들을 피벗을 기준으로 분할 할 수 없을 때 까지 수행한다.

- **BurbleSort**
  a. 인접한 두 요소를 선택한다.
  b. 내림차순/오름차순 정렬의 방법에 따라, 두 요소를 비교 한 후 정렬 기준에 맞지 않으면 서로 교환한다. 따라서 한 번 모든 요소를 비교 할 때 마다 가장 마지막 index에 가장 작은 값/가장 큰 값이 오는 것이 보장된다.

2.3 알고리즘 적용 서비스

키워드를 기반으로 여행 정보가 담긴 모든 테이블에 join을 이용한 sql 문으로 해당 키워드가 설명이나 여행지 이름에 포함되어 있을 경우 뽑아내 List 형태로 받아준다.
이 때 받은 List 형태의 파라미터를 jsp 페이지에서 받아 각각의 정렬 알고리즘으로 정렬한 후, 순서대로 화면에 출력해준다. request에 List가 담겨있으므로, jsp 페이지 내에서 스크립틀릿을 이용해 java 코드로 정렬을 수행했다.
정렬 알고리즘마다 가지고 있는 안정성은 해당 기능에서 영향을 주지 않기 때문에, 정렬 알고리즘 중 구현이 비교적 어렵지 않고 평균적으로 좋은 성능을 내는 QuickSort와, 구현은 정렬 중 제일 쉽지만 성능이 좋지 안은 BurbleSort 두 알고리즘을 선택해 정렬을 수행했다.

```jsp
<%!
  public static void quickSort(bannerDto[] arr, int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);

        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}

public static int partition(bannerDto[] arr, int low, int high) {
    bannerDto pivot = arr[high];
    int i = (low - 1);

    for (int j = low; j < high; j++) {
        if (arr[j].getReadCount() >= pivot.getReadCount()) {
            i++;

            bannerDto temp = arr[i];
            arr[i] = arr[j];
            arr[j] = temp;
        }
    }

    bannerDto temp = arr[i + 1];
    arr[i + 1] = arr[high];
    arr[high] = temp;

    return i + 1;
}
%>
```

Figure 7: QuickSort를 적용

```
bannerDto[] burArr = lst.toArray(new bannerDto[lst.size()]);
bannerDto[] quickArr = lst.toArray(new bannerDto[lst.size()]);
System.out.println(lst.size() + " 리스트 사이즈");
// 퀵 정렬
long startTimeQuick =  System.currentTimeMillis();
quickSort(quickArr, 0, quickArr.length - 1);
long endTimeTim = System.currentTimeMillis();

long startTimeBubble = System.currentTimeMillis();
// 버블 정렬
for (int i = 0; i < burArr.length; i++) {
    for (int j = i; j < burArr.length; j++) {
        int result1 = burArr[i].getReadCount();
        int result2 = burArr[j].getReadCount();

        // 더 많이 읽은 수로 내림차순 정렬
        if (result1 < result2) {
            bannerDto tmp = burArr[i];
            burArr[i] = burArr[j];
            burArr[j] = tmp;
        }
    }
}
long endTimeBurble = System.currentTimeMillis();
long elapsedTime = endTimeBurble - startTimeBubble;
long elapsedTime2 = endTimeTim - startTimeQuick;
boolean burchk = true;
if (!burchk) System.out.println("버블 정렬 실패!");
boolean quickchk = true;
for (int i = 1; i < burArr.length; i++) {
    if (quickArr[i - 1].getReadCount() < quickArr[i].getReadCount
        quickchk = false;
        break;
    }
}
if (!quickchk) System.out.println("퀵 정렬 실패!");
System.out.println("버블 정렬 시간 : " + elapsedTime);
System.out.println("퀵 정렬 시간 : " + elapsedTime2);
request.setAttribute("list", quickArr);
```

Figure 8: BurbleSort & QuickSort를 호출해서 사용