웹 해킹이란 무엇인가?

Web Hacking Tutorial

패스워드 보안(MD5, SHA256)

[BOSS] 손 우 규

https://github.com/swk3169/web-hacking

목차

1. 패스워드 보안(MD5, SHA256)

1.1 정의

MD5(Message-Digest algorithm 5)는 128비트 암호화 해시 함수이다. RFC 1321로 지정되어 있으며, 주로 프로그램이나 파일이 원본 그대로인지를 확인하는 무결성 검사 등에 사용된다. 1991년에 로널드 라이베스트가 예전에 쓰이던 MD4를 대체하기 위해 고안했다.

1996년에 MD5의 설계상 결함이 발견되었다. 이것은 매우 치명적인 결함은 아니었지만, 암호학자들은 해시 용도로 SHA-1과 같이 다른 안전한 알고리즘을 사용할 것을 권장하기 시작했다. 2004년에는 더욱 심한 암호화 결함이 발견되었고. 2006년에는 노트북 컴퓨터 한 대의 계산 능력으로 1분 내에 해시 충돌을 찾을 정도로 빠른 알고리즘이 발표되기도 하였다. 현재는 MD5 알고리즘을 보안 관련 용도로 쓰는 것은 권장하지 않으며, 심각한 보안 문제를 야기할 수도 있다. 2008년 12월에는 MD5의 결함을 이용해 SSL 인증서를 변조하는 것이 가능하다는 것이 발표되었다.

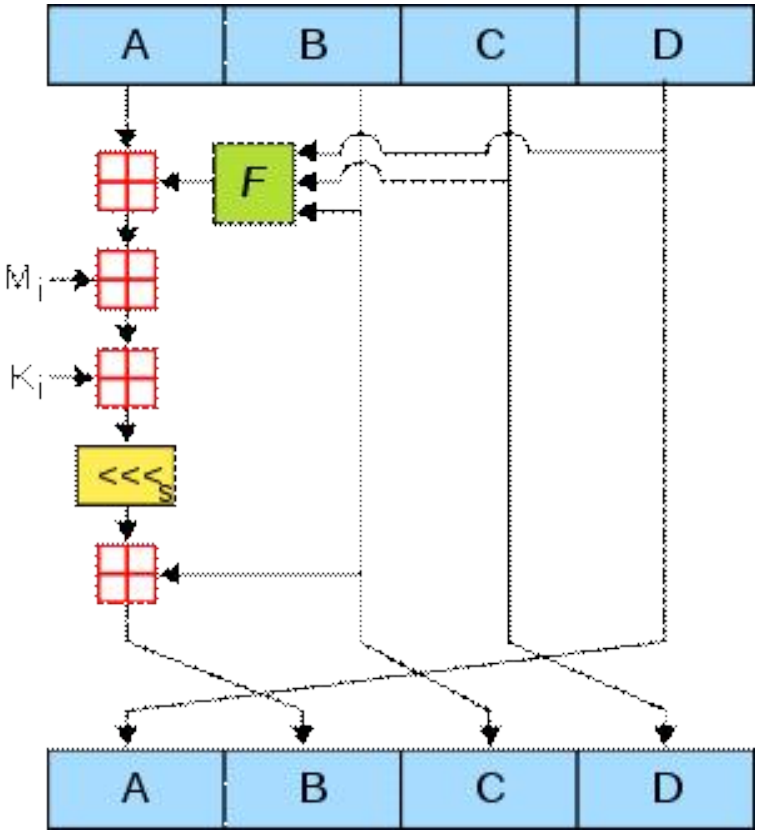
1.2 방어법

1.2.1 MD5(Message-Digest algorithm 5)

MD5는 임의의 길이의 메시지(variable-length message)를 입력받아, 128비트 짜리 고정 길이의 출력값을 낸다. 입력 메시지는 512 비트 블록들로 쪼개진다; 메시지를 우선 패딩하여 512로 나누어떨어질 수 있는 길이가 되게 한다. 패딩은 다음과 같이 한다: 우선 첫 단일 비트, 1을 메시지 끝부분에 추가한다. 512의 배수의 길이보다 64 비트가 적은 곳까지 0으로 채운다. 나머지 64 비트는 최초의(오리지널) 메시지의 길이를 나타내는 64 비트 정수(integer)값으로 채워진다.

메인 MD5 알고리즘은 A,B,C,D라고 이름이 붙은 32 비트 워드 네 개로 이루어진 하나의 128 비트 스테이트(state)에 대해 동작한다. A,B,C,D는 소정의 상수값으로 초기화된다. 메인 MD5 알고리즘은 각각의 512 비트짜리 입력 메시지 블록에 대해 차례로 동작한다. 각 512 비트 입력 메시지 블록을 처리하고 나면 128 비트 스테이트(state)의 값이 변하게 된다.

하나의 메시지 블록을 처리하는 것은 4 단계로 나뉜다. 한 단계를 "라운드"(round)라고 부른다; 각 라운드는 비선형 함수 F, 모듈라 덧셈, 레프트 로테이션(left rotation)에 기반한 16개의 동일 연산(similar operations)으로 이루어져 있다. 아래 그림은 한 라운드에서 이루어지는 한 연산(operation)을 묘사하고 있다.

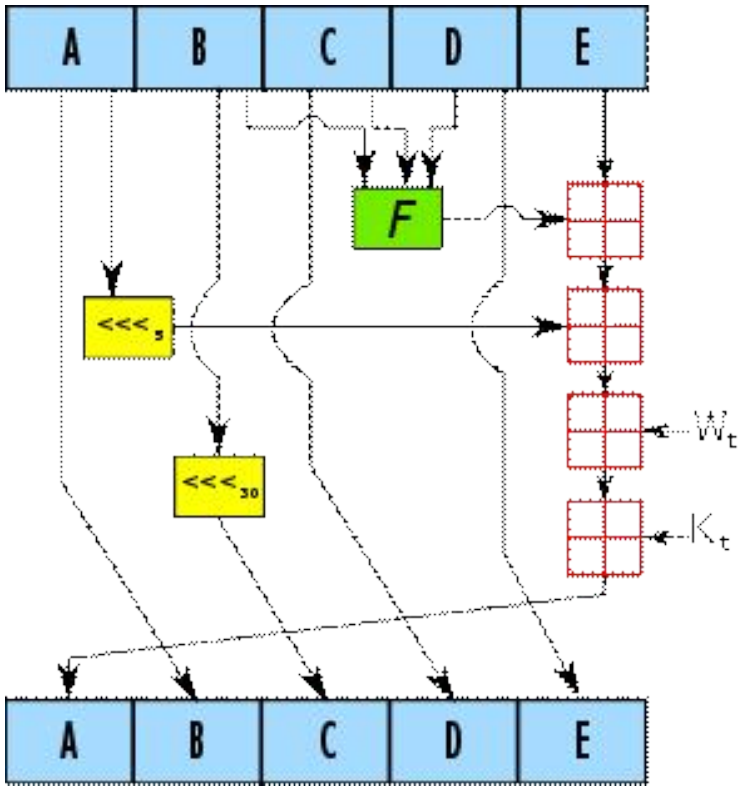


단일 MD5 연산. MD5에서는 이 단일 연산을 64번 실행한다. 16개의 연산을 그룹화한 4 라운드로 묶인다. F는 각 라운드에서 사용하는 비선형 함수를 가리키며, 각 라운드에서는 각각 다른 함수를 사용한다. Mi는 입력 메시지의 32-비트 블록을 의미한다.
left shifts는 s칸 만큼의 레프트 로테이션을 가리키며, s는 각 연산 후 값이 변한다. Addition 은 모듈로 232 덧셈을 말한다.

1.2.2 SHA256(Secure Hash Algorithm)

최초의 알고리즘은 1993년에 미국 표준 기술 연구소(NIST)에 의해 안전한 해시 표준(Secure Hash Standard, FIPS PUB 180)으로 출판되었으며, 다른 함수들과 구별하려 보통 SHA-0이라고 부른다. 얼마 안 있어 NSA는 이 표준을 폐기했고, 1995년에 개정된 알고리즘(FIPS PUB 180-1)을 새로 출판했으며 이를 SHA-1이라고 부른다. SHA-1은 SHA-0의 압축 함수에 비트 회전 연산을 하나 추가한 것으로, NSA에 따르면 이는 원래 알고리즘에서 암호학적 보안을 감소시키는 문제점을 고친 것이라고 하지만 실제로 어떤 문제점이 있었는지는 공개하지 않았다. 일반적으로 SHA-1은 SHA-0보다 암호학적 공격이 힘든 것으로 알려져 있으며, 따라서 NSA의 주장은 어느 정도 설득력이 있다. SHA-0과 SHA-1은 최대 264비트의 메시지로부터 160비트의 해시값을 만들어 내며, 로널드 라이베스트가 MD4 및 MD5 해시 함수에서 사용했던 것과 비슷한 방법에 기초한다.

NIST는 나중에 해시값의 길이가 더 긴 네 개의 변형을 발표했으며, 이들을 통칭하여 SHA-2라 부른다. SHA-256, SHA-384, SHA-512는 2001년에 초안으로 처음으로 발표되었으며, 2002년에 SHA-1과 함께 정식 표준(FIPS PUB 180-2)으로 지정되었다. 2004년 2월에 삼중 DES의 키 길이에 맞춰 해시값 길이를 조정한 SHA-224가 표준에 추가되었다. SHA-256과 SHA-512는 각각 32바이트 및 64바이트 워드를 사용하는 해시 함수이며, 몇몇 상수들이 다르긴 하지만 그 구조는 라운드의 수를 빼고는 완전히 같다. SHA-224와 SHA-384는 서로 다른 초기값을 가지고 계산한 SHA-256과 SHA-512 해시값을 최종 해시값 길이에 맞춰 잘라낸 것이다.



SHA-1 압축 함수가 블록 하나를 처리하는 과정. A, B, C, D, E는 각각 32비트 내부 상태이고, F는 계속 변하는 비선형 함수이며, Kt는 상수이다. 왼쪽 회전n은 n비트만큼 왼쪽으로 회전하는 연산이고, 덧셈은 232 모듈로 덧셈을 나타낸다.

1.3 문제점

이 두 가지의 암호화 모듈은 전형적인 단방향 해시 함수의 문제점을 갖는다.

1.3.1 Brute force attack

brute force는 그 단어의 느낌처럼 무식하게 조합 가능한 모든 패스워드를 대입해보는 것이다. 대입해서 똑같은 해시값이 나오면 빙고. 기본적으로 모든 해시는 brute force로 뚫을 수 있다. 다만, 그 시간이 엄청나게 오래 걸리게 만들어서 뚫기 어렵게 만들 수 있을 뿐이다. md5가 보안 목적으로 적합하지 않은 것은 일단 기본적인 brute force로도 꽤 빠른 시간에 뚫린다는 것이다.

1.3.2 Rainbow table

그럼 만약에 해시 알고리즘을 엄청나게 복잡하게 만들어서 오래 걸리게 하면 brute force 공격으로부터 안전해지지 않을까? 이를테면 SHA-512의 경우는 앞서의 공격으로도 초당 364000개 밖에 대입하지 못한다고 한다. 물론 이것도 엄청나지만 앞서의 8자리 영문자 숫자 조합의 경우 7만년 쯤 걸린다. 이 정도면 안전하지 않을까?

그렇지 않다. Brute force attack을 가만히 생각해보면 더 좋은 방법을 생각해낼 수 있다. 미리 가능한 패스워드 조합을 다 계산한 테이블을 가지고 비교만 수행하는 것이다. 이것이 dictionary attack인데, 이 dictionary를 해시값 검색에 최적화시킨 것을 rainbow table이라고 한다. md5의 경우는 인터넷에 이미 수백억 개의 해시값에 대한 rainbow table이 있다. 이미 계산된 값을 이용하므로 알고리즘의 복잡도도 큰 상관이 없다. 이 안에 있는 패스워드는 그냥 금방 뚫리는 것이다. 공개된 md5 rainbow table로 찾으면 대부분의 웹사이트에서 90% 정도의 사용자 패스워드가 크랙된다고 하니 이쯤되면 md5는 이미 뚫려 있다.

물론 rainbow table을 만드는데도 시간이 많이 들기 때문에 알고리즘의 복잡도가 전혀 상관이 없는 것은 아니다. 알고리즘 수행시간이 길면 충분한 규모의 rainbow table을 확보하기도 어렵다. 그래도 쉬운 패스워드는 뚫리니까 보안성을 크게 높일 수 있다고 말하긴 어렵다.

1.3.3 Collision attack

하지만 사실 md5가 broken 판정을 받은 것은 brute force 때문이 아니다. md5는 1996년에 이미 collision 취약점에 대한 이론적인 가능성이 제시되었고, 이후에 그게 현실로 나타났다. collision은 서로 다른 두 원본 메세지가 같은 해시값을 갖는 경우를 말한다. 해시 함수의 특성상 collision은 존재할 수 밖에 없는데, 이런 collision을 찾기 힘든 특성을 collision resistance라고 하며 이것은

암호화 목적의 해시에 필수 요소다. 그런데, md5는 그 collision resistance가 낮아서 collision을 쉽게 찾을 수 있는 것이다.

그런데, 여기에 또 반전이 있다. md5가 collision attack에 취약해서 암호화 용도 폐기 판정을 받았지만, 사실 패스워드 암호화는 collision attack과 상관이 없다. collision attack으로 같은 해시값을 갖는 두 개의 데이터를 만들어내기는 쉽지만, 단순히 해시값이 주어졌을 때 그 해시값에 맞는 collision들을 찾아낼 수 있는 것은 아니다. 그래서, collision attack은 패스워드 크랙이 아니라 인증서 위조, 악성코드가 담긴 실행파일 만들기 등에 사용된다.

1.3.4 Preimage attack

해시 알고리즘을 패스워드 암호화에 쓸 수 있는지 판단하는 기준은 preimage attack인데, 이것은 해시값에서부터 원본 데이터를 찾아낼 수 있는 가능성이다. 이건 md5를 포함한 대부분의 해시 함수가 안전하기 때문에 크게 걱정할 필요는 없으나, 암호학을 모르는 사람이 직접 만든 해시 함수를 사용하지 말아야 하는 이유가 되긴 한다.

1.4 해결방안

collision attack과 preimage attack은 별로 걱정할 필요가 없다면 패스워드 보안에서 중요한 건 brute force와 rainbow table에 대한 방어다. 그래서 이에 대한 대처는 두 가지로 요약된다.

- 암호화 시간을 많이 걸리게 만들어서 brute force 공격의 효율성을 떨어뜨린다.
- salt를 이용해서 rainbow table을 무의미하게 만든다.

1.4.1 Salt

해시에 대한 다양한 크랙 수단이 있지만, 그 중 가장 효율적인 크랙 수단은 물론 rainbow table이다. 그런데, 이 rainbow table은 또한 아주 쉽게 무력화할 수 있는 방법이기도 하다. 단순히 동일한 salt값을 추가하는 것만으로도 미리 계산해놓은 rainbow table은 꽤 힘이 빠진다. 예를 들어 패스워드를 codeok라고 정했는데 단순히 md5로만 해싱했다면 다음과 같은 값이 나온다.

8587229bbf6f4dc5efa906f04291519f

위의 해시값이 rainbow table에 들어 있다면 이 패스워드의 계정은 바로 크랙

된다. 물론, rainbow table에 위의 해시값으로 매핑된 원본 텍스트가 codeok는 아닐 수 있다. collision이 있기 때문이다. 하지만 패스워드 체크는 같은 값만 나오면 되기 때문에 통과할 수 있다.

그런데, 만일 패스워드를 암호화할 때 salt로 .net을 붙여서 codeok.net을 md5로 해싱했다고 해보자. 그러면 다음과 같은 값이 나온다.

25f43a96006d7a7b1434384e8acc6f5e

이 값으로 rainbow table을 검색했을 때 나온 원본 텍스트를 비밀번호 입력창에 넣는다면 어떻게 될까? 그러면 그 원본 텍스트에 또 .net을 붙여서 암호화한 다음 비교하기 때문에 실패하게 된다. 설령 원본 텍스트로 codeok.net이 나왔다고 해도 소용 없다. 그래서 단순한 고정값 salt로도 rainbow table의 힘을 떨어뜨릴 수 있다.

그러나, 고정값 salt만 있으면 rainbow table에서 salt까지 포함한 테이블을 만들 수도 있다. 예를 들어서 salt 길이가 1자라고 한다면, rainbow table을 만들 때 패스워드 허용 문자 개수를 곱한 만큼 만들면 된다. 수십 배 커지지만 상대적으로 엄청난 rainbow table의 사이즈에 비하면 크게 늘어난다고 하기는 어렵다. 그래서 salt는 최소 128 bit 정도는 되어야 안전하다고 한다.

근데 이게 끝이 아니다. 설령 충분히 긴 salt를 주더라도 rainbow table에 당할 수 있다. 많은 수의 패스워드를 rainbow table로 찾아서 원본 텍스트를 나열해 보면 공통적인 부분이 발견될 수 있기 때문이다. 만일 위의 두번째 해시값에서 원본 텍스트로 codeok.net이 나왔고, 또 다른 해시값에서 newsqu.net이 나왔다면 .net이 salt라고 추정할 수 있다. 그래서 salt 값도 고정값을 쓰면 안되고 암호학적으로 안전한 랜덤 함수를 이용하는 것이 좋다.

salt의 목적이 rainbow table의 무력화이므로 salt 값은 그냥 패스워드 해시값과 같이 저장해도 상관 없다. Django의 경우 패스워드 해시와 salt를 한 필드에 같이 저장한다. 왠지 같이 저장하면 해싱을 reproduce할 수 있기 때문에 안전하지 않을 것 같은 느낌이 들지만, 어차피 패스워드 암호화는 reproduce해야 하고, salt는 rainbow table만 막으면 된다.

1.4.2 알고리즘 수행 시간

salt로 rainbow table을 막았다면 이제 남은 것은 brute force 공격이다. 반복하지만, brute force는 원천적으로 막을 방법은 없고, 암호화 알고리즘을 느리게

만들어서 brute force 공격의 효율을 떨어뜨리는 방법 뿐이다. 그렇다면 엄청나게 복잡하게 느린 알고리즘을 사용하면 되겠네. 땡!

더 좋은 답은 알고리즘 수행 시간을 조정 가능한 방법을 쓰는 것이다. 컴퓨팅 파워는 계속 향상되고, 필요한 보안 수준은 사이트에 따라, 사이트의 성장에 따라 달라진다. 알고리즘 수행 시간을 조정하는 가장 쉬운 방법은 해싱을 반복하는 것이다. 간혹 md5로 암호화를 해놓고, 좀더 보안 수준을 높여보겠다면서 sha1(md5(password)) 같은 식으로 하는 경우가 있다. 이것도 의도는 좋으나, 해싱을 두 번 반복하는 것 뿐이므로 brute force 공격 앞에서는 별다른 의미가 없다. PBKDF2 같은 경우는 반복회수를 조정할 수 있는데, 권장하는 최소 반복회수가 1000번이다.

1.4.3 올바른 패스워드 저장 방식

패스워드 크랙을 대비하기 위해 salt도 써야 하고, 해싱을 반복까지 해야 하다니 할일이 제법 많다. 하지만 이걸 직접 짜라는 게 아니다. 귀찮은 일이기도 하거니와, Home grown crypto is bad crypto이기 때문이다. 위의 내용을 다 포괄하고 있는 암호화 알고리즘들이 이미 옛날옛적에 개발되어 있다.

1.4.4 사용자 인터페이스의 고려사항

패스워드를 저장만 잘한다고 되는 게 아니다. 서두에서 인터페이스만 보고도 보안 문제가 의심되는 경우가 있다고 했는데, 지켜야 할 것이 몇 가지가 있다. 우선 패스워드의 개수를 제한하는 것은 되도록 삼가해야 한다. 특히 아주 작은 값, 예를 들어 10이나 12 정도로 패스워드를 제한하고 있으면 이놈들 패스워드를 평문으로 저장하고 있는 것 아닌가? 하는 의심이 든다. 실제로 과거의 프로그래머들은 데이터베이스의 컬럼 길이를 되도록 작게 잡으려는 경향이 있었고, 그 컬럼 크기에 맞춰서 패스워드 길이를 제한했었기 때문이다. 그런데 패스워드를 제대로 해싱한다면 패스워드가 아무리 길어도 해싱하면 거의 고정 길이의 텍스트가 나오기 때문에 저장에 문제가 없다. 그래서 패스워드 길이 제한을 할 필요가 없는데도 작은 값으로 제한하고 있다면 평문 저장이 아닌지 의심하게 되는 것이다.

물론 최소 길이는 제한하는 것이 좋다. 짧은 패스워드는 brute force로 빨리 잡힐 수 있기 때문이다.

길이 제한이 나쁜 것은 평문 저장이 의심되기 때문만은 아니다. 길이 제한을 알게 되면 brute force의 범위를 좁힐 수 있기 때문이다. 앞서 md5의 사례에서 보듯, 비밀번호가 8자 정도로 제한되면 아주 빠른 시간에 뚫릴 수 있다. 물론

pbkdf2나 bcrypt를 사용한다면 훨씬 안전하지만, 그래도 되도록 길이를 얼마로 제한하고 있는지 모르게 하는 것이 좋다.

—

2. 실습

2.1 [Node.js] md5, sha256 모듈을 이용한 비밀번호 보안

)nodejs의 모듈을 사용하여 비밀번호를 보안하는 방법에 대해서 알아보겠습니다.

```
C:\Users\swk31\dev\js\server_side_javascript>
C:\Users\swk31\dev\js\server_side_javascript>npm install md5 --save
```

npm의 md5 모듈을 설치합니다.

```
var md5 = require('md5');
```

모듈을 다음과 같이 불러옵니다.

```
app.post('/auth/login', function(req, res){
    var uname = req.body.username;
    var pwd = req.body.password;
    for(var i=0; i<users.length; i++){
        var user = users[i];
        if(uname === user.username && md5(pwd+user.salt) === user.password){
            req.session.displayName = user.displayName;
            return req.session.save(function(){
                res.redirect('/welcome');
            });
        }
    }
    res.send('Who are you? <a href="/auth/login">login</a>');
});
```

로그인 시 기존 비밀번호와 비교할 새롭게 입력된 비밀번호에 모듈을 사용합니다. ex) `md5(pwd+user.salt)`

```
+ md5@2.2.1
updated 1 package in 2.515s

C:\Users\swk31\dev\js\server_side_javascript>node
> var md5 = require('md5');
undefined
> md5('111');
'698d51a19d8a121ce581499d7b701668'
```

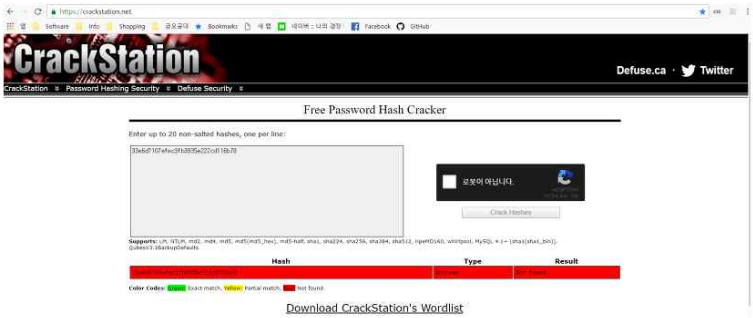
모듈이 설치된 것을 확인, node를 실행하여 암호화 할 숫자를 다음과 같이 모듈과 함께 선언합니다. ex) 암호화 할 숫자 111 -> md5('111');

```
var users = [
    {
        username: 'swk',
        password: '698d51a19d8a121ce581499d7b701668',
        salt:'!@#!@##Aaa',
        displayName:'swk'
    },
    {
        username: 'K8805',
        password: '698d51a19d8a121ce581499d7b701668',
        salt:'!@$!$@#$',
        displayName:'K5'
    }
];
```

password에 암호화 된 코드를 선언합니다. 각각의 다른 salt로 인하여 외부에서는 다르지만 내부적으로는 같은 비밀번호를 갖게 됩니다.

로그인과 로그아웃이 잘 이루어진 것을 확인할 수 있었습니다.

하지만 md5는 해커들에 의해서 취약점이 쉽게 노출될 수 있는 모듈입니다.



https://crackstation.net/

다음과 같은 사이트에서 암호화 된 코드를 쉽게 변환하여 복호화할 수 있습니다.

md5 모듈을 이용한 비밀번호 보안

전체 코드

그리하여 다음과 같이 더 복잡한 암호화를 할 수 있는 sha256 모듈을 사용해보겠습니다.

```
C:\Users\swk31\dev\js\server_side_javascript>npm install sha256 --save
```

모듈을 설치합니다.

```
var sha256 = require('sha256');
```

모듈을 선언합니다.

```
app.post('/auth/login', function(req, res){
    var uname = req.body.username;
    var pwd = req.body.password;
    for(var i=0; i<users.length; i++){
        var user = users[i];
        if(uname === user.username && sha256(pwd+user.salt) === user.password){
            req.session.displayName = user.displayName;
            return req.session.save(function(){
                res.redirect('/welcome');
            });
        }
    }
    res.send('Who are you? <a href="/auth/login">login</a>');
});
```

로그인 시 기존 비밀번호와 비교할 새롭게 입력된 비밀번호에 모듈을 사용합니다. `sha256(pwd+user.salt)`

```
var users = [
    {
        username: 'swk',
        password: '111',
        salt:'!@#!@##Aaa',
        displayName:'swk'
    },
    {
        username: 'K8805',
        password: '111',
        salt:'!@$!$@#$',
        displayName:'K5'
    }
];
```

각 salt의 값을 확인

```
> var sha256 = require('sha256');
undefined
> var pwd = '111';
undefined
> var salt = '!@#!@##Aaa';
undefined
> sha256(pwd+salt)
'c5b5a9ed37dfee64f9879de97482c432bce6655daf6c2680ff64504c28f7696e'
```

사용자의 비밀번호 값과 salt의 값을 더하여 sha256 모듈을 사용합니다.

```
> var salt = '!@$!$@#$';
undefined
> sha256(pwd+salt);
'd6ede6c095c85166adf234d8731d129ad6067685a27b991038453179ebf4a990'
>
```

마찬가지로 더한 값을 확인합니다.

```
var users = [
    {
        username: 'swk',
        password: 'c5b5a9ed37dfee64f9879de97482c432bce6655daf6c2680ff64504c28f7696e',
        salt:'!@#!@##Aaa',
        displayName:'swk'
    },
    {
        username: 'K8805',
        password: 'd6ede6c095c85166adf234d8731d129ad6067685a27b991038453179ebf4a990',
        salt:'!@$!$@#$',
        displayName:'K5'
    }
];
```

salt와 pwd가 더해진 값을 password에 입력

로그인이 잘 이루어지는 것을 확인합니다.

sha256 모듈은 md5보다 더 복잡하게 암호화된 코드를 가질 수 있으며

이를 기존의 패스워드 값과 더하게 되면 더 복잡한 암호화 코드를 생성할 수 있습니다.

그리고 이러한 모듈을 해시 함수라고 부릅니다.

3. 참조

- http://www.codeok.net/%ED%8C%A8%EC%8A%A4%EC%9B%8C%EB%93%9C%20%EB%B3%B4%EC%95%88%EC%9D%98%20%EA%B8%B0%EC%88%A0