Store and validate hashed password

1 “用户名+密码”认证简介

采用“用户名+密码”这种登录方式的系统如何实现认证呢？

最原始的方法是用户注册后，系统把用户的密码保存到数据库（或文件）中；当用户登录时，把用户的输入密码和之前系统保存的密码对比，两者相同就通过认证，不同就拒绝认证。这种办法的缺点在于，如果攻击者得到了保存用户密码的数据库，则用户密码就直接被攻击者获取了。显然，直接明文保存用户密码的作法是非常不安全的。

一种容易想到的改进思路是：不直接保存用户密码，而是保存用户密码的 hash 值（比如说密码对应的 MD5 值）。当用户登时，把用户的输入密码进行相同的 hash 运算后得到 hash 值后与数据库中保存的 hash 值对比，两者相同就通过认证，不同拒绝认证。这样，就算攻击者得到了保存用户密码 hash 值的数据库，他也无法通过 hash 值反推出用户的密码。但是，这种法也是不安全的。一般来说，用户密码不会设置得太长，假设用户的密码是不超过 8 位的数字或字母，攻击者可以把不超过位的数字或字母的所有可能组合提前计算出 hash 值保存到一个表中（称为 Rainbow Table，即“彩虹表”），这样攻击者得到存用户密码 hash 值的数据库后，就可以直接查询彩虹表，从而得到用户的真正密码。

2 防止“彩虹表攻击”：计算 hash 前加 salt

为了防止使用“彩虹表”批量破解哈希值，我们可以采用对密码加盐 salt（一个随机字符串）后再计算 hash 值的方式。具体过如下所述（注：下面的介绍仅是原理性的，不要直接拿来用到生产系统中）。

假设系统采用的 hash 算法为 MD5，用户 user1 在注册时指定密码为 8 位字符“Lq!bZ/05”，我们生成一个 15 位随机数“!x.%1Qp^3q,8hQz”作为盐值，则我们把密码和盐值的组合“!x.%1Qp^3q,8hQzLq!bZ/05”的 MD5 值 d9ace71d953f73d3a37ba9d3ab1cd30d 保存到数据库中。又假设用户 user2 在注册时指定密码为 8 位字符“64pXi.rW”，我生成一个 15 位随机数“s2A1ksI*72fe#41”作为盐值，则可算出密码和盐值的组合“s2A1ksI*72fe#4164pXi.rW”对应 MD5 值 f4b0cc569f41f9330f200f3bcdfb30b5。为了能够正确地验证密码，我们必须需要把盐值也保存到数据库中。

系统用户数据库如下所示：

user	salt	hash
user1	!x.%1Qp^3q,8hQz	d9ace71d953f73d3a37ba9d3ab1cd30d
user2	s2A1ksI*72fe#41	f4b0cc569f41f9330f200f3bcdfb30b5

攻击者通过非法手段获得了系统的数据库后，通过查询“彩虹表”，一般情况下找不到 d9ace71d953f73d3a37ba9d3ab1cd3和 f4b0cc569f41f9330f200f3bcdfb30b5 的对应记录。因为它们是 15+8=23 位字符的 hash 值，一般来说，MD5 的“彩虹表”能覆盖任意 1 到 10 位字符的 hash 值就很不错了，目前还没有能覆盖任意 23 位字符的 MD5“彩虹表”（因为这个数据量在太大了）。如果我们把 salt 的长度变得更长，则通过查询“彩虹表”来破解 MD5 就更加不可行了。

这时，攻击者只能采用“暴力破解”方法了。比如，想要破解用户 user1 的密码，攻击者得计算字符串“!x.%1Qp^3q,8hQzXXXXXXXX”（其中 XXXXXXXX 是未知的任意组合）的 MD5 值，一旦发现其中某个字符串（就是“!x.%1Qp^3q,8hQzLq!bZ/05”）的 MD5 值为 d9ace71d953f73d3a37ba9d3ab1cd30d 时，则破解出了用户 user1 的密码（为“Lq!bZ/05”）。

通过上面的分析可知，要实现较好的安全性，salt 需要满足：

1. salt 不能太短。假设 salt 很短（比如为 2 个字符），则利用“彩虹表”很可能可以直接破解 hash。
2. salt 要足够随机，不能重复使用（特别地，每个用户的 salt 要不同；同一个用户在修改密码后，salt 也要更换）。假设系中有 1 万个用户，且使用的 salt 都相同，尽管破解某一个指定用户密码的难度不变，但攻击者破解所有用户密码需要尝的次数将大大减少，所以使用相同 salt 会大大降低了“暴力破解”的难度。

3 防止“哈希长度扩展攻击”：不要使用 hash(salt || password)方式，请使用 HMAC

hash(salt || password)这种方式（其中||表示字符串连接）可以防止“彩虹表”攻击，但它也不安全。利用一种称为“哈希长度扩
攻击(Length extension attack)”的手段可能非法进入系统。

Hash functions like MD5, SHA1, and SHA2 use the Merkle–Damgård construction, which makes them vulnerable to wha known as length extension attacks. This means that given a hash Hash(key + message), an attacker can compute Hash(pad(key + message) + extension), without knowing the key.

下面通过一个例子（摘自：哈希长度扩展攻击, 绿盟科技博客）来演示哈希长度扩展攻击。

假设 web 系统的密码验证逻辑为：

```
$auth = false;
if (isset($_COOKIE["auth"])) {
  $auth = unserialize($_COOKIE["auth"]);
  $hsh = $_COOKIE["hsh"];
  if ($hsh !== md5($SECRET . strrev($_COOKIE["auth"]))) {    //$SECRET is a 8-bit salt
    $auth = false;
  }
} else {
  $auth = false;
  $s = serialize($auth);
  setcookie("auth", $s);
  setcookie("hsh", md5($SECRET . strrev($s)));
}
```

为了方便，我们用下面类似的 python (example.py)来测试：

```
import hashlib
import sys
from urllib import unquote

def login(password, hash_val):
    m = hashlib.md5()
    secret_key = "message"          # secret_key is a salt
    m.update(secret_key + password)
    if(m.hexdigest() == hash_val):
        print "Login Successful!"
    else:
        print "Login Failed"

if __name__ == "__main__":
    password = unquote(sys.argv[1])
    hash_val = unquote(sys.argv[2])
    login(password, hash_val)
```

已知系统中用户 user1 的密码为 root（攻击者不知道），对应的 salt 为 message（攻击者不知道），md5(salt || password)=md5(messageroot)=f3c36e01c874865bc081e4ae7af037ea（攻击者通过非法途径攻取了）。

```
$ python example.py root f3c36e01c874865bc081e4ae7af037ea
Login Successful!
```

利用“哈希长度扩展攻击”（不详细介绍原理和步骤），我们可以算出一个密码和相应 hash 值，它们可以通过上面的密码验
辑，测试如下：

```
$ python example.py
%80%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%0
 e53a681a30ff99e3f6522270ca7db244
Login Successful!
```

通过上面的输出知，攻击者使用伪造的密码和 hash 值进入了系统。

需要说明的是，“哈希长度扩展攻击”需要应用具备下面条件：

1. 使用 hash(key || message)这种方式，且使用了 MD5 或 SHA-1 等基于 Merkle–Damgård 构造的哈希函数生成哈希值；
2. 让攻击者可以提交数据以及哈希值，虽然攻击者不知道密文；
3. 服务器把提交的数据跟密文构造成字符串，并经过哈希后判断是否等同于提交上来的哈希值。

在前面的攻击实例中，服务器接受用户输入的密码和 hash 值。不过，在用户登录过程中，可设计服务器仅接受一个密码，算密码相应 hash，再和数据库中的值对比，即可实现认证过程（这个过程并不会受到“哈希长度扩展攻击”）。

参考：
哈希长度扩展攻击的简介以及HashPump安装使用方法
哈希长度扩展攻击解析
科普:哈希长度扩展攻击

3.1 Keyed-Hash Message Authentication Code (HMAC)

前面介绍过，利用 md5(key || message)方式（其中||表示字符串连接）生成的 hash 可能会受到“哈希长度扩展攻击”。

那么，我们可以使用 md5(message || key)或者 md5(md5(key) || message)等方式生成 hash 来避免“哈希长度扩展攻击”吗？答是：不要这么做，请使用密码专家设计的方案——Keyed-Hash Message Authentication Code (HMAC)，其 RFC 文档为：https://tools.ietf.org/html/rfc2104

HMAC 仅是一种混合 key 和 message 的方法，可以使用任意 hash 函数，如使用 md5/sha1/sha256 等 hash 算法时，相应 HMAC 算法可称为 HmacMD5/HmacSHA1/HmacSHA256。

HMAC 的伪代码实现为：

```
function hmac (key, message) {
  if (length(key) > blocksize) {
    key = hash(key) // keys longer than blocksize are shortened
  }
  if (length(key) < blocksize) {
    // keys shorter than blocksize are zero-padded (where ∥ is concatenation)
    key = key ∥ [0x00 * (blocksize - length(key))] // Where * is repetition.
  }

  o_key_pad = [0x5c * blocksize] ^ key // Where blocksize is that of the underlying hash function
  i_key_pad = [0x36 * blocksize] ^ key // Where ^ is exclusive or (XOR)

  return hash(o_key_pad ∥ hash(i_key_pad ∥ message)) // Where ∥ is concatenation
}
```

3.1.1 HMAC 实例（openssl, Java）

可以直接使用工具 openssl 来测试 hmac，如下：

```
$ echo -n "password" | openssl dgst -sha1 -hmac "salt"
c1d0e06998305903ac76f589bbd6d4b61a670ba6
$ echo -n "data" | openssl dgst -sha1 -hmac "key"
104152c5bfdca07bc633eebd46199f0255c9f49d
```

Java 中测试如下：

```
// from https://gist.github.com/ishikawa/88599/3195bdeecabeb38aa62872ab61877aefa6aef89e
import java.security.InvalidKeyException;
import java.security.NoSuchAlgorithmException;
import java.security.SignatureException;
import java.util.Formatter;

import javax.crypto.Mac;
import javax.crypto.spec.SecretKeySpec;


public class HmacSha1Signature {
    private static final String HMAC_SHA1_ALGORITHM = "HmacSHA1";

    private static String toHexString(byte[] bytes) {
        Formatter formatter = new Formatter();

        for (byte b : bytes) {
            formatter.format("%02x", b);
        }

        return formatter.toString();
    }

    public static String calculateRFC2104HMAC(String data, String key)
        throws SignatureException, NoSuchAlgorithmException, InvalidKeyException
    {
        SecretKeySpec signingKey = new SecretKeySpec(key.getBytes(), HMAC_SHA1_ALGORITHM);
        Mac mac = Mac.getInstance(HMAC_SHA1_ALGORITHM);
        mac.init(signingKey);
        return toHexString(mac.doFinal(data.getBytes()));
    }

    public static void main(String[] args) throws Exception {
        String hmac = calculateRFC2104HMAC("data", "key");

        System.out.println(hmac);
        assert hmac.equals("104152c5bfdca07bc633eebd46199f0255c9f49d");
    }
}
```

4 让暴力破解更慢：使用"Slow Hash Function"

现代计算机的计算能力越来越强大，还可以利用 GPU 加速，使得计算字符串的 MD5，SHA1 等 hash 值非常快，这使得暴解需要的时间越来越短。

为了更好地抵抗暴力破解，我们可以设计 hash 算法，使得计算一次 hash 值需要的时间比较长，从而使暴力破解的时间变长这类慢 hash 算法有：PBKDF2, bcrypt, scrypt, Argon2, crypt（$2y$, 5, 6 版本）等等。

说明：不要使用 MD5, SHA1, crypt（1, 2, $2x$, 3 版本）等对密码进行 hash 运算，它们抵抗暴力破解的能力很差。

5 总结：如何保存和验证密码

To Store a Password

1. Generate a long random salt using a Cryptographically Secure Pseudo-Random Number Generator.
2. Prepend the salt to the password and hash it with a standard **password hashing function** like Argon2, bcrypt, scrypt, or PBKDF2, etc.
3. Save both the salt and the hash in the user's database record.

To Validate a Password

1. Retrieve the user's salt and hash from the database.
2. Prepend the salt to the given password and hash it using the same hash function.
3. Compare the hash of the given password with the hash from the database. If they match, the password is correct. Otherwise, the password is incorrect.

参考：Salted Password Hashing - Doing it Right

6 Unix crypt function

Unix-like 系统中的 crypt 函数，是一个可以加 salt（只能是两位字符）的 hash 函数。以前用它来加密系统用户的密码。

下面是函数 crypt 的简单测试实例：

```
$ cat test.c
#define _XOPEN_SOURCE
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int main() {
   printf("%s\n", crypt("password", "sa"));
   exit(0);
}
$ gcc test.c -lcrypt -o test
$ ./test
sa3tHJ3/KuYvI
```

很多其它语言也实现了 crypt，测试如下：

```
$ python -c 'import crypt; print crypt.crypt("password","sa")'
sa3tHJ3/KuYvI
$ ruby -e 'print "password".crypt("sa"); print("\n");'
sa3tHJ3/KuYvI
$ perl -le "print crypt('password','sa');"
sa3tHJ3/KuYvI
```

参考：
https://en.wikipedia.org/wiki/Crypt_(C)
man 3 crypt

6.1 Modular Crypt Format

原始的 crypt 函数早已不再安全，现代 Unix-like 系统不使用原始的 crypt 来加密用户密码。

现代系统中，一般通过扩展 salt 的格式来支持其他 hash 算法，扩展后的 salt 格式称为 Modular Crypt Format 。

Modular Crypt Format 的基本格式（使用 $ 作为分隔符）为：

```
$id$salt$encrypted
```

下面是 Modular Crypt Format 的一些例子：

Schema Id	Schema	Example
	DES	sa3tHJ3/KuYvI
1	MD5	1etNnh7FA$OlM7eljE/B7F1J4XYNnk81
2,2a,2x,2y	bcrypt	$2a$10$VIhIOofSMqgdGlL4wzE//e.77dAQGqntF/1dT7bqCrVtquInWy2qi
3	NTHASH	$3$$8846f7eaee8fb117ad06bdd830b7586c
5	SHA-256	$5$9ks3nNEqv31FX.F$gdEoLFsCRsn/WRN3wxUnzfeZLoooVlzeF4WjLomTRFD
6	SHA-512	6qoE2letU$wWPRl.PVczjzeMVgjiA8LLy2nOyZbf7Amj3qLIL978o18gbMySdKZ7uepq9tmMQ

Table 1: Modular Crypt Format

例如，我的 Linux 中，保存用户密码使用的是 crypt 6 。

```
$ sudo cat /etc/shadow
......
cig01:$6$gL5SlRI4$eez954NDwizilXE6jIQ9VfzeYmL3UigUIcDLLlPm.eAXVv/i3XdNdhSlO97cOTgzSRkXL93kP1JZxQ6/0XLbQ1:16620:0:99999:7:::
......
```

用户 cig01 的密码为 12345（不要使用这样的简单密码），使用 python 验证如下：

```
$ python -c 'import crypt; print crypt.crypt("12345", "$6$gL5SlRI4$")'
$6$gL5SlRI4$eez954NDwizilXE6jIQ9VfzeYmL3UigUIcDLLlPm.eAXVv/i3XdNdhSlO97cOTgzSRkXL93kP1JZxQ6/0XLbQ1
```

上面输出和/etc/shadow中保存的信息一致，从而验证了cig01的密码确实为12345。