A short introduction to password security

Franco Raimondi
BIS4610 - Information Security Management
Week 11

Middlesex University

By the end of today...

You should be

- able to articulate the issues associated with password management;
- familiar with the mechanism of hash functions;
- understand the notion of *salting*;
- able to articulate the steps involved in dictionary and brute force attacks;

Motivation

- Passwords are the most common mechanism to provide *authentication*.
- A number of security issues are associated with password storage and transmission: security managers need to be aware of these.
- The controls associated to password storage and transmission may be subject to regulations (e.g., for PCI-DSS compliance:
  - *8.4 Render all passwords unreadable during storage and transmission, for all system components, by using strong cryptography.*

Example of incidents

- 6.4M Linkedin passwords (hashed): `http://mashable.com/2012/06/08/linkedin-stolen-passwords-list/`
- 8.4M Gamigo username and password (hashed): `http://www.zdnet.com/8-24-million-gamigo-passwords-leaked-after-hack`
- 450k Yahoo! passwords leaked *in clear text*: `http://arstechnica.com/security/2012/07/yahoo-service-hacked/`
- 100k IEEE passwords leaked *in clear text*: `http://ieeelog.com/`

Some interesting statistics from Yahoo! and Linkedin

See: `http://blog.eset.se/`
`statistics-about-yahoo-leak-of-450-000-plain-text-a`
and `http://mashable.com/2012/06/08/`
`linkedin-stolen-passwords-list/`

- The most common passwords include: `123456`, `password`, `welcome`, `ninja`, `abc123`, `qwerty`, `link`, etc.
- Two lessons to learn here:
    1. People still use very easy passwords (think about enforcing strong password in your organisation)
    2. Do not assume that organisations store passwords in a secure way (Yahoo! and IEEE passwords are clear text): thus, *do not re-use the same password!*

Hash functions

Hash function = an algorithm that takes some data (of variable length) and generates a fixed-length *hash value* (also known as *message digest*, or *digest*), such that:

- It is computationally easy to compute the digest of a message.
- It is computationally infeasible to obtain the original message from the digest.
- A small change in the original message causes a big change in the digest.
- It is computationally infeasible to find to messages with the same digest.

Applications of hash functions

- We have used hash functions to verify the integrity of messages in *digital signatures*: compute the digest of a message, encrypt with private key, and send to recipient.
- Hash functions can be used to verify the integrity of files (e.g., files downloaded form Internet and disk images: see BIS4605 and BIS4615!).

For passwords: the most common mechanism is to store the *hash value* of the password, instead of the clear-text password.

Some practical examples

- Common hash functions: MD5 (128 bit output), SHA-1 (160 bit output), SHA-256/224 (256/224 bit output), SHA-512/384 (512/384 bit output).
- From Linux command line:

```
$ echo -n "Hello World" | md5sum
  b10a8db164e0754105b7a99be72e3fe5
$ echo -n "Hello World" | sha1sum
  0a4d55a8d778e5022fab701977c5d840bbc486d0
```

  (and so on with sha256sum and sha512sum).
- Various tools are available on-line if you want to play a bit with hash functions.

Storing passwords as a list of hashes

Passwords are typically stored hashed. Is this OK? It depends...
This is a list of password hashes (using SHA-1):

```
fdf8bc5814536f66012884e146a8887a44709a56
87acec17cd9dcd20a716cc2cf67417b71c8a7016
b1b3773a05c0ed0176787a4f1574ff0075f7521e
[...]
```

How would you proceed to find the corresponding passwords?

Approach 1

(The wrong one)

1. generate all combinations of number, letters (upper and lower case) and symbols,
2. compute the digest of each one of them,
3. check if it is in the list.

It is computationally (and physically) impossible to explore a 128 bit state space, see `http://en.wikipedia.org/wiki/Brute_force_attack#Theoretical_limits`

Approach 2

As seen above, a number of users employ simple passwords. Suppose you have a list of common words: it can be used to perform a *dictionary attack*:

1. Pre-compute the digest of all the words in the dictionary.
2. For each hash value in the file, check if it is in the list of “known” hashes.

Is this feasible? Yes!

- 6.4M hashes; a dictionary with 900k words: 10 seconds to find approx 85K weak passwords on an average laptop!

Quick demo

- Compute the SHA-1 hash of some of your passwords.
- Send the passwords to me.
- We will try to see if they are in my dictionary (I use a simple python program, approx 20 lines of code).

Salted passwords

- The attack described above is very efficient because we can *pre-compute* the hash value of the words in the dictionary.
- To make this attack more difficult, it is possible to add a unique value to the password before computing its hash value. Example: for user `john` with password `abc123`, we store the value of `SHA-1("johnabc123")`.
- The unique value is called a *salt*.
- In reality: the salt is a random sequence and is stored (in clear) together with the result of the hashing operation.

Example: /etc/shadow in Unix systems

```
franco:$6$z78kdhje$Kl/mqoe728ajxiGTHH7[...]:15307:0:99999:7:::
john:$6$ieoak7.jSk$JSH289sjqk/8jKlmUqj[...]:15459:0:99999:7:::
```

Fields are separated by “:” and the format is:

- User name
- `$id$salt$hash`, where id=6 means SHA-512
- Date of last pwd change (as days from 1/1/1970), and then other numbers related to valid from, expiry, etc.

Salted passwords and dictionary attacks

Are dictionary attacks feasible with salted passwords?

- Yes, but they take longer. For a list of 1000 salt+hash and a dictionary of 1M words, we need to compute 1 billion hashes (and we cannot re-use them): for each password in our dictionary, we need to compute the salted hash using a different salt for each user!
- Additional complication: the hashing procedure could be iterated more than once before storing the password, possibly thousands of times!
- Still feasible, but **much slower**: 300 hashes and 900K passwords take 6 hours 30 min on the same laptop above (compare with 10 seconds for 6.4M unsalted hashes!).
- In a production environment, 9 out of 300 passwords could be found with a (small) dictionary of 900K words, so the attack is still effective.

A useful tool: John the ripper

`http://www.openwall.com/john/`
A free tool to detect weak passwords. It can be used in various ways (brute force or wordlist). It recognises automatically the salt etc. Given a (readable) shadow.txt file and a wordlist.txt file, just run:

```
$ john --wordlist=wordlist.txt shadow.txt
```

JTR is highly configurable and customisable. One possibility is to use *rules* to *mangle* a list of words. Possible rules are:

```
p pluralize: "crack" -> "cracks", etc. (lowercase only)
P "crack" -> "cracked", etc. (lowercase only)
I "crack" -> "cracking", etc. (lowercase only)
S shift case: "Crack96" -> "cRACK(^"
V lowercase vowels, uppercase consonants: "Crack96" -> "CRaCK96"
```

Additional material: storing password in memory

Developers need to be careful when manipulating sensitive data. A C example:

```
int validate(char *username) {
  char *password;
  char *checksum;
  password = read_password();
  digest = compute_digest(password);
  erase(password);  /* securely erase password */
  return !strcmp(digest, get_stored_digest(username));
}
```

(material taken from
https://www.securecoding.cert.org/confluence/display/seccode/MSC18-C.+Be+careful+while+handling+sensitive+data,+such+as+passwords,+in+program+code
and from
http://www.cgisecurity.com/lib/protecting-sensitive-data.html)

Conclusion

- Passwords should never be stored or transmitted in clear.
- When stored, password should be salted and hashed.
- Passwords should not be easy to guess; organisations should check that the passwords employed by users are not subject to dictionary attacks.
- Passwords should not be re-used across different applications / web sites (as there is no guarantee on how they are manipulated).

Useful links

- NIST SP 800-63 “Electronic Authentication Guideline”, `http://www.nist.gov/manuscript-publication-search.cfm?pub_id=910006`
- John the Ripper `http://www.openwall.com/john/`
- PCI-DSS `https://www.pcisecuritystandards.org/`