Cryptographic Hashes

Peter Norris

October 2016

Contents

Preamble

Starting assumptions

1: Almost none, specifically, you have:

2: a Linux account but little experience of using the Linux command shell;

3: a logical mind but little experience of logic and number systems;

4: the ability to experiment, take accurate notes, and learn rapidly from both your successes and failures;

5: a questioning attitude (including the recognition that this file contains deliberate errors; copy + paste is your enemy to learning)

Intended outcomes

6: Can explain the properties of cryptographic hashes.

7: Can apply appropriate hash techniques to a range of problems.

8: Can use a small range of Linux commands to automate the application of cryptographic hashes.

Typographic conventions

9: Commands, to be entered at the shell prompt will be as shown below:

```
10: sha1sum * | tea hashes-that-i-saved.txt
```

11: Commands that need some customised input from you will be represented by:

```
12: md5ssum -c <the-filename-you-used-earlier>
```

13: Directives for action will typically be in bold. Some of these will require you to figure out *how* to do things. Anything you do not instinctively already know, **make sure you add to your notes** for the session.

14: Prompts for questions you should pause and ask of yourself will be italicised. *Why should I have to pause when I have done what was asked?*

Preliminaries

15: **Launch a shell.**

16: The shell provides an interface for you to define the instructions you want to be executed. The shell takes the line of characters that you type and, when you hit *return*, figures out what needs to be done, does it, and lets you have the interactive shell prompt back when it has finished.

17: In addition to doing the obvious things, the shell will also do a range of more subtle things: variable substitution, character expansion, file redirection and so forth. We will use a few of these to make things more efficient.

18: **Create a directory** to contain the cryptographic hashing work you are about to undertake.

```
19:  mkdir -p ~/crypto/hash
```

20: What does -p do?

21: What does ~ represent?

22: **Put yourself in the directory** you just created:

```
23:  cd ~/crypto/hash
24:  pwd
```

25: As ever, use the **man** command to find out what a particular command does.

Hash concepts

26: A cryptographic hash function produces a fixed length summary output (the digest) from an input sequence of 1s and 0s of arbitrary length (the message). A good cryptographic hash function has the following properties:

27: a) it ***always*** produces the same digest from the same message (it is therefore deterministic),

28: b) it is ***not feasible*** to create two different messages which produce the same digest (it is therefore a reliable discriminator),

29: c) it is ***not feasible*** to derive the message from the digest (it is a one way function - easy to run forwards, infeasible to run backwards)

30: When the same digest is produced by two different messages, a collision has occurred. A collision is so unlikely that when one is detected, the hash function is regarded as "broken".

Hash applications

31: Applications of cryptographic hash functions are almost entirely concerned with establishing message integrity.

32: Storing passwords,

33: Confirming a file has not been altered over time,

34: Confirming a file has not been altered in transit,

35: Detecting that a file fails to match a known "good" file,

36: Detecting that a file matches a known "bad" file,

37: Generally, confirming that the content of two files is (not) identical.

Passwords (and password cracking)

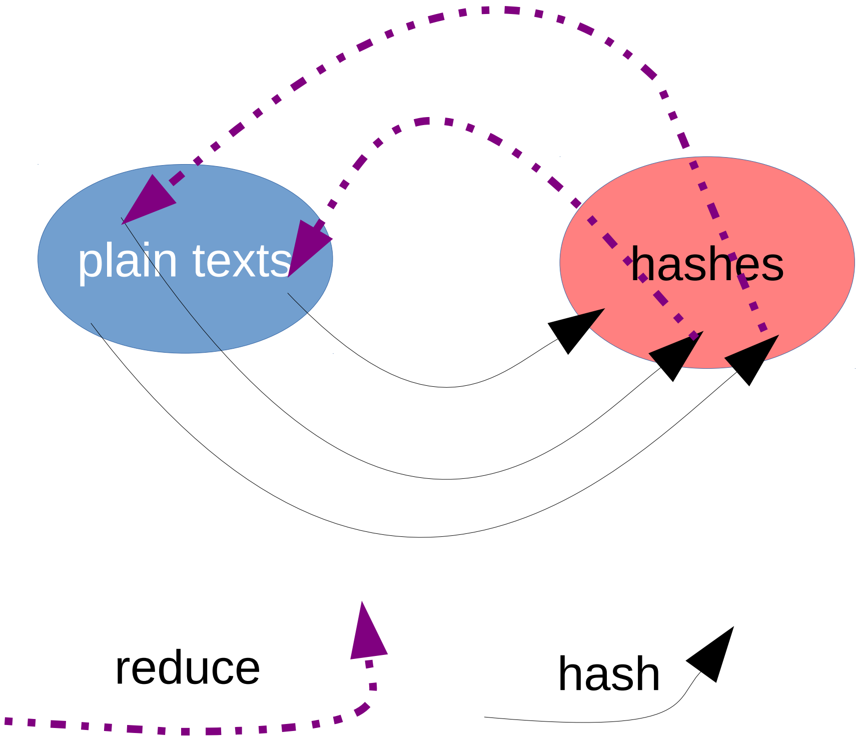
38: **Storing passwords in clear text is not wise**. Anyone gaining access to the the password file is able to read the passwords.

39: However, **if the hash of the password is stored**, then anyone gaining access to the password file is prevented from reading the passwords. Authentication is achieved by comparing the hash of the password which the user entered with the the hash of the correct password, stored in the password file. If the hashes match, then the user supplied the right password. If they do not match, the user supplied the wrong password.

40: However, even though the hash itself is not reversible, the propensity of people to choose passwords from a small dictionary subset of the (almost) infinite variety of available passwords can be exploited. **Hashes of dictionary words** can be pre-calculated and stored in carefully ordered / indexed hash tables. If the hash of an unknown password is presented to a hash table, then the corresponding clear-text password can be quickly looked up (provided that it existed in the original dictionary from which the hash table was constructed).

41: However, hash tables of **all possible passwords** (rather than just those from a dictionary), would occupy enormous amounts of storage. Just assume a user were to choose a password which was a **random combination** of up to 20 characters from the ASCII keyboard (upper and lower case alphabet, numeric digits and punctuation gives approx 80 different characters), there would be:
$80^{20} = 115{,}292{,}150{,}460{,}684{,}697{,}600{,}000{,}000{,}000{,}000{,}000{,}000 \approx 10^{38}$.
different passwords available.

42: However, you can trade off computing time for storage space in pre-calculated and stored in carefully indexed **rainbow tables**. The following diagram shows roughly what is going on for a three link rainbow table:



43: Think of the "reduce" function as a way of converting an arbitrary hash into an arbitrary plaintext. A given hash will always reduce to the same plaintext BUT that plaintext is different from the one we would put through the hash function to get the original hash.

44: All that is stored in the rainbow table is many random plaintexts and for each plaintext, the corresponding final hash, resulting from a given number of hash, reduce, hash, reduce, hash, …, reduce, hash cycles.

45: If the hash of an unknown password is presented to a rainbow table, then the corresponding clear-text password can be iteratively found.

46: Rainbow tables can be defeated (ie made to costly to produce and store) by storing the hash of a ***salted*** password. The authentication system stores two items: a random message (the salt) and the hash of the salt concatenated to the password. A password presented for authentication is first prefixed with the salt, then hashed. If the hash matches the stored hash, then the submitted password is authentic. Long random salting of passwords increases the size of the set of likely password hashes so much, that the size of the rainbow tables becomes too great to be useful.