Why I Like Cascaded Hash Password Generation

A caveat the reader should be aware of is that in what follows, I have worked according to what is easy for *me* to remember, rather than an average user. Suppose we have the truncated hash output:

```
9HeKoSyUmK4XDOju
```

Simple Salt+String hashing

Consider first the situation where we know that this is of the form "SaltSite" where "Salt" is some meaningful sentence (MyCatIsBobbles) and "Site" is the name of the site, possibly with username, e.g. twitter@MrBobbles.

The problem we need to solve, given the hash output above, is to find an input to SHA256 (assuming we know it is SHA256) of the form "SomeSentenceSiteStuff", for example "MyCatIsBobblestwitter@MrBobbles".

This may be amenable to brute forcing using a dictionary.

Hash Cascading

Consider now, however, if we do the following:

hash("My".hash("Cat".hash("Is".hash("Bobbles".hash("twitter@MrBobbles")))))

Now, assuming we know the wordlist, [My,Cat,Is,Bobbles], we know that the input to the final (leftmost) hash is of the form "MyXYZ1", where XYZ1 is the output of sha256 where the input was of the form "CatXYZ2", where XYZ2 is the output of sha256 where the input was of the form "Is", etc. That said, all it does is slow down brute force by a linear factor (the number of words).

Hash Cascading with Modifications

If we give ourselves just one more independent piece of information, what then? Now if instead we replace the word "Is" with something else, possibly the output of another sha256 hash, then suddenly the attacker will need to brute force this correctly in order to get the final output correctly. If this one extra bit is independent of the main input, an attacker must get both "twitter@MrBobbles" and what we replace "Is" correct. This replacement is easily done with a sed command (and replacing the nth word is easily done with an awk command). In the example attack problem that follows, we restrict ourselves to just a sed command.

The Attack Problem

Assuming that they know:

1. That this process is an iterated hash construct, using a word list like [My,Cat,Is,Bobbles];

2. That this word list is commonly known, and for definiteness assume that it is a <10-verse passage from the KJV or the Latin Vulgate, all punctuation removed, all words made lowercase.

3. They know this word list has been put through an arbitrary but memorable sed command (e.g.

    sed -e “s/In/bouncy/g” -e “s/.*/& & &/” -e “s/bouncy/Flump/”

or simply

    sed -e “s/thy/MyHamster/”

which is amenable to memorisation, and things like

    sed -e “s/the/$(echo Turnip | sha256sum)/”

are allowed, though if you wish, also consider possibilities when such things are not allowed).

4. That the main input is exactly of the form like “twitter@MrBobbles”;

5. That the hash output is converted to base64 and truncated to 16 characters;

6. They have the output of this process for the user @MrBobbles on twitter; and

7. They know that “twitter@MrBobbles” is the input.

Given the above, how hard is it to work out the arguments passed to sed?

Example

An easier variant:

If I were, specifically, I take the word list (John 1:1 from the Latin Vulgate) “in principio erat verbum et verbum erat apud deum et deus erat verbum”, put it through:

```
sed -e “s/.*/&&&/” -e “s/erat/XYZZY/g”
```

where XYZZY is some secret string (e.g. BomblesTheFluffcat, and you may assume for now that it matches the pattern /([A-Z][a-z]{1,16}){1,5}/), and &&& is a string matching the pattern /&[&]+&/.

Knowing that the input is

    “twitter@MrBobbles”

and the output is

“9HeKoSyUmK4XDOju”

how hard is it to work out what &&& and XYZZY actually are?

(for those who can't read sed, -e “s/.*/& & & &/” takes the initial word string, duplicates it four times, and -e “s/erat/BomblesTheFluffcat/g” replaces every “erat” with “BomblesTheFluffcat”)

Ease of Implementation

Note that, given a standard Linux command line, one can do this cascaded hashing with the following script:

```
#!/bin/bash

hasher() {
    # $1 is state, $2 is new word
    echo -n "$1$2" | sha256sum | cut -c1-64 | xxd -r -p
}
X="$(hasher "$1" "")"; shift
if [ -n "$*" ]; then
    WORDS=(`<words.txt sed "$@"`)
else
    WORDS=(`<words.txt`)
fi
for a in "${WORDS[@]}"; do
    X="$(hasher "$X" "$a")"; shift
done
OUTPUT="$(echo -n "$X" | base64)"
echo "${OUTPUT:0:16}"
```

and for anybody reasonably familiar with bash scripting and the standard command line tools used, this is easy enough to reconstruct from memory.

Importantly, if:

1. the word list can be reconstructed from memory or recovered with rudimentary internet access

2. you can remember the modifications, and

3. you can write a functionally equivalent script to the above, and

4. all you have is a standard Linux bash prompt, possibly on a machine that is not yours,

you can get your passwords back.

The idea I am exploring is simultaneously maximising password security and minimising memorisation effort and requiring no pre-written custom software or stored data, merely a working bash prompt and standard GNU utilities (bash, sed, sha256sum, etc., as one will also get on Windows with their 'linux subsystem') and *whatever you can carry in your memory*.

Alas gnu sed does not have a 'change nth word' function, but this could easily be done in advance with

```
awk: <words1.txt awk '{$7=”HexVision”}' >words2.txt
```