Find a collision on a weakened SHA-1

HW2 - CNS Sapienza

Andrea Fioraldi 1692419

2018-11-6

1 Introduction

Cryptographic hash functions play a key role in the modern information security world. They are used for browser security, for fingerprinting of data and in code repositories. SHA-1 is a popular hashing function designed in 1995 at NSA. It is (or was we hope) widely used in popular applications like GIT and also in SSL certificates.

In this report, I will discuss how to find a collision on a weakened version of SHA-1 defined as $extract(SHA1(message), 104, 160)$, that is the last 56 bits of the output.

The code is written entirely in C and the program was run on a laptop with an Intel I7-7700hq processor, 16 gigabytes of RAM and an SSD.

2 Design

In order to find such collision, I exploited the fact that SHA-1 has the 2^n pre-image resistance property, but not the $2^{n/2}$ collision resistance. Thanks to the birthday paradox it is possible to generate $2^{n/2}$ (where $n = 56$ in our case) hashes from different inputs and have a high probability that two of the generated hashes are the same.

An high-level algorithm is presented using the following pseudocode:

```
ht = new HashTable
for i = 0 to ATTEMPTS_NUM
  m = unique_random_string()
  o = weakened_sha1(m)
  if o in ht
```

```
    collision_found(m, ht[o])
    exit()
  ht[o] = m
```

3 Implementation

I choose to set `ATTEMPTS_NUM` to $2 * 2^{n/2}$, so $2 * 2^28$, that is the double of the birthday bound so a reasonable number of attempts.

A first problem is how the inputs messages for SHA-1 must be generated. I decided to generate random text and not random bytes, in order to produce two human-readable strings as inputs related to the collided hash. To do that I cannot use the regular random number generator of the C standard library but I implemented a generator that ensures the uniqueness of the produced random string. I chose to start from a variant of UUID to generate a unique random 64-bit number N and then use it to generate a 16 bytes long string. The characters of such string are in the range $['a','p']$ because each character of index i is defined as $'a' + extract(N, i * 4, i * 4 + 4)$. The UUID variant code is based on an old implementation made by Apple in 2003 [1].

Another problem concern memory usage. The hash table can be larger than the memory of the laptop. I tried a simple implementation using the C++ `std::map` class and I failed. So then I considered using a hash table data structure that works with the disk. I found an open source implementation on GitHub [2] and I modified a bit the code to make the structure works with a bytes array as the key instead of a C string like in the original implementation because the weakened hash can contain bytes 0.

A space optimization that I added to the algorithm is to use the index i as the object stored in the hash table instead of the full message. Saving the initial seed allows us to reconstruct the two messages after that the collision is found. This optimization is a trade-off between the hash table size and the time spent in the calculation. The index i can be stored in the hash table as an array of 4 bytes that is much better than store the full input of 16 bytes.

To compute the SHA-1 we used the API offered by libcrypto.

The code of both the disk and RAM versions can be found at [3].

4 Results

Running the program that I developed to find a collision on the last 56 bits of SHA-1 you can get a collision after circa one hour using a similar hardware. Probably with a computer with more RAM the time would be very smaller but thanks to the SSD this time is reasonable. Using a hard disk is not practicable.

I found the collision after 457064599 attempts. Here the output generated by the program:

```
andrea@malweisse ~/hw2
$ time ./collision
seed: 1541565841
progress: 85.134888 %  [457064448 of 536870912]

COLLISION!!!
idexes: 457064599   118325688
data: mfjfmlafgdjehahb
hash: bb4e8711dded28f7d2705f4e30efa0ca596a67ec
data: dmfhijjodlmnomnb
hash: a1f274649d69e5836502bf7924efa0ca596a67ec
./collision  660,92s user 776,71s system 37% cpu 1:03:58,07 total
```

The final result is that $extract(SHA1(\text{"}mfjfmlafgdjehahb\text{"}), 104, 160) = extract(SHA1(\text{"}dmfhijjodlmnomnb\text{"}), 104, 160)$.

5 Conclusion

I chose to consider only 56 bits as proof of work of my code because of the little amount of time I could leave the PC working [1].

As a future work, the idea and its implementation can be extended to works with more bits. The first optimization may simply use the RAM of a powerful instance in the cloud instead of the laptop with the SSD. Secondly, we can parallelize the hash computation in several cores of the CPU using synchronized access to a shared queue.

A lesson remains: I will never use SHA-1 again for security!

[1] this week there was the Cyber Security Awareness Worldwide and I was in France for the associated academic hacker teams competition

References

[1] *uuid.c - Apple CoreFoundation 299.35.* `https://opensource.apple.com/source/CF/CF-299.35/Base.subproj/uuid.c.auto.html` Accessed: 2018-11-7.

[2] *luispedro/diskhash: Diskbased (persistent) hashtable - GitHub.* `https://github.com/luispedro/diskhash` Accessed: 2018-12-11.

[3] *andreafioraldi/weakened-sha1-collision: Find a collision on a weakened version of SHA-1 - GitHub.* `https://github.com/andreafioraldi/weakened-sha1-collision` Accessed: 2018-12-7 (avaiable only after the homework deadline).