x) Read and summarize (with some bullet points)

Schneier 2015: Applied Cryptography: Chapter 2 - Protocol Building Blocks: subchapters "2.3 One-way Fuctions" and "2.4 One-Way Hash Functions".

- According to the chapter 2.3 One-way Functions (Schneier, 2015) the one way functions are very easy to compute, but very hard to reverse.
- One-way function is central to public-key cryptography
- A message encrypted with the one-way function isn't useful as no one could decrypt it
- A trapdoor one-way function is a function that is easy to compute in one direction and hard to compute in the other direction.
- This read was interesting in a way that I have several use cases behind where hash functions have been used (related to data integrity, passwords & certificates), but without knowledge of the concepts of “trapdoor one-way function”, “one-way functions” or their differences.

Karvinen 2022: Cracking Passwords with Hashcat

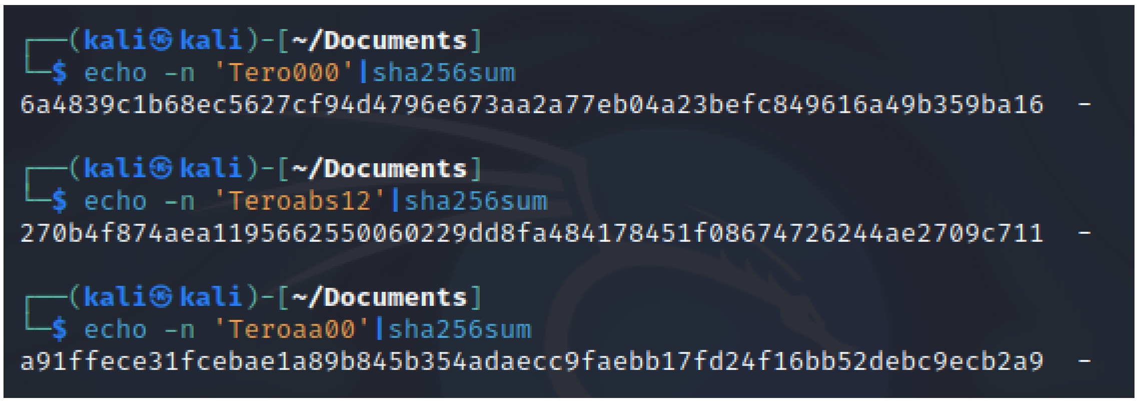
- The article describes in detail how to use hashcat tool for cracking passwords
- It came to me as a new thing that you can use “hashid -m” command to identify hash type. I’m not professionally inclined in a way that cracking hashes would be a normal job for me (even though hashcat is a familiar tool and I have used it before to some extent) so this kind of detail was a nice new thing to know.
- There’s also tip that you can boost hashcats performance with running it on your host OS and utilizing your display adapter for it.

Karvinen 2020: Command Line Basics Revisited

- This article goes through the basics when working with command line and working and moving between directories, file manipulation etc.
- The article is a comprehensive cheat sheet for anyone to educate (or re-educate, when you’re working in those legacy servers in production you really REALLY want to get the commands right) themselves on command line basics.

a) Billion dollar busywork.

As a first try I tried to add stuff to the string without any automation and picking the “stuff” (numbers and strings) randomly. Some of the commands I put in without the wanted result:

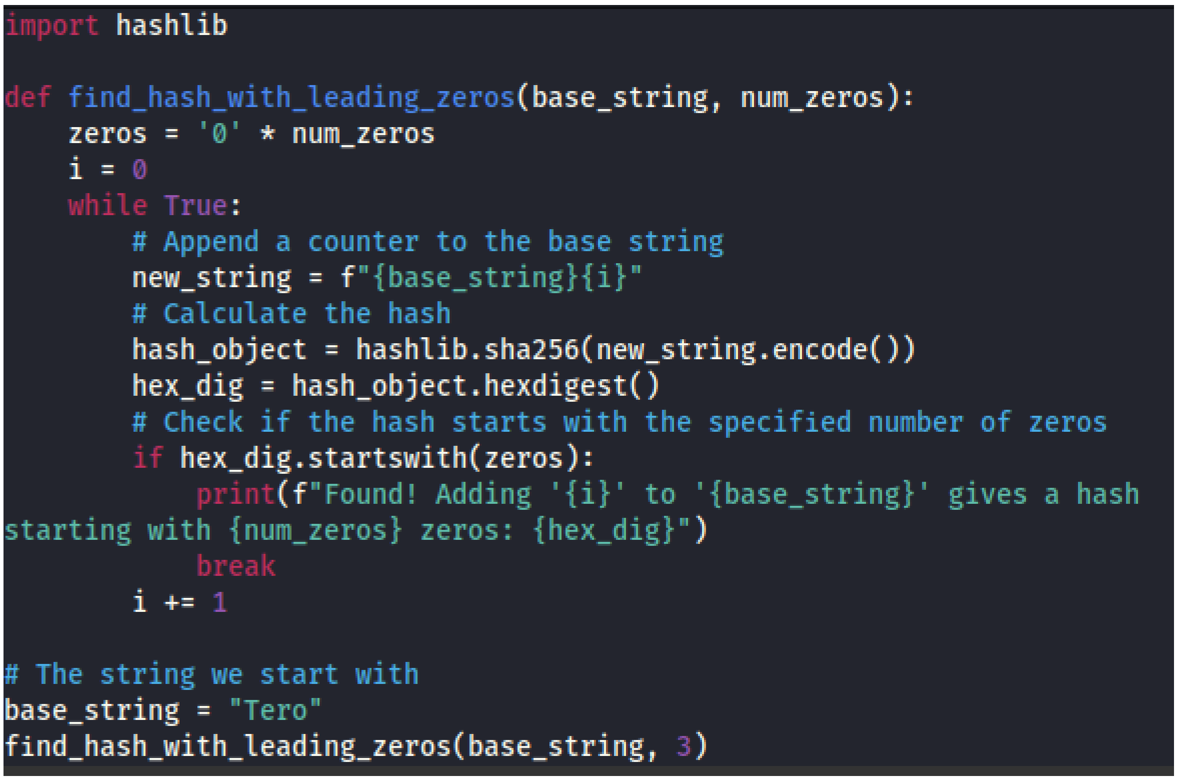


```
┌──(kali㉿kali)-[~/Documents]
└─$ echo -n 'Tero000'|sha256sum
6a4839c1b68ec5627cf94d4796e673aa2a77eb04a23befc849616a49b359ba16  -

┌──(kali㉿kali)-[~/Documents]
└─$ echo -n 'Teroabs12'|sha256sum
270b4f874aea1195662550060229dd8fa484178451f08674726244ae2709c711  -

┌──(kali㉿kali)-[~/Documents]
└─$ echo -n 'Teroaa00'|sha256sum
a91ffece31fcebae1a89b845b354adaecc9faebb17fd24f16bb52debc9ecb2a9  -
```

After that I started to think how this could be automated and I asked ChatGPT to make a python script that would add three zeros to the hash (three straight away because you only live once). The script that ChatGPT produced:

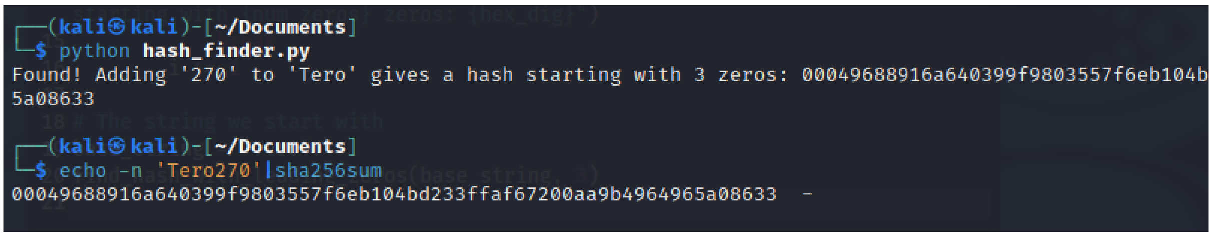


```python
import hashlib

def find_hash_with_leading_zeros(base_string, num_zeros):
    zeros = '0' * num_zeros
    i = 0
    while True:
        # Append a counter to the base string
        new_string = f"{base_string}{i}"
        # Calculate the hash
        hash_object = hashlib.sha256(new_string.encode())
        hex_dig = hash_object.hexdigest()
        # Check if the hash starts with the specified number of zeros
        if hex_dig.startswith(zeros):
            print(f"Found! Adding '{i}' to '{base_string}' gives a hash 
starting with {num_zeros} zeros: {hex_dig}")
            break
        i += 1

# The string we start with
base_string = "Tero"
find_hash_with_leading_zeros(base_string, 3)
```

And when I run it I got the following result:

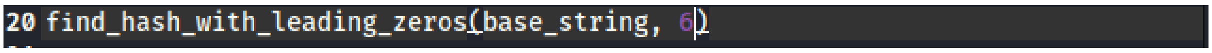


```
┌──(kali㉿kali)-[~/Documents]
└─$ python hash_finder.py
Found! Adding '270' to 'Tero' gives a hash starting with 3 zeros: 00049688916a640399f9803557f6eb104b
5a08633

┌──(kali㉿kali)-[~/Documents]
└─$ echo -n 'Tero270'|sha256sum
00049688916a640399f9803557f6eb104bd233ffaf67200aa9b4964965a08633  -
```

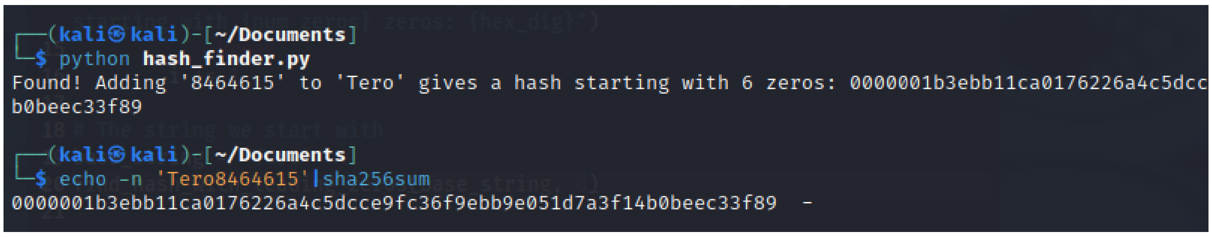
When studying the subject, prompting the ChatGPT and analyzing results there's two things I learned about the matter. First is that just adding random letters and numbers to the string in order to add zeros in the beginning by trial and error is tedious task for a human. At least for a human who knows only so much about the topic. This kind of task is by principle computationally intensive and is more easy to solve with computational logic and automation. The second thing I learned pretty fast when I was learning this topic from internet is that this task is linked to bitcoin mining closely as the bitcoin miners are trying to find hash functions that start with certain amounts of zeros defined by Bitcoin network's consensus rule.

By changing this one line of code (bonus points to ChatGPT for producing such maintainable code) I got six zeros and I guess you could add even more (as long as you have the computing power) with this script. I tried to run it a bit to produce 12 zeros, but my hacking laptop started to sound like it's on the last mile of a marathon and I had to move forward so I stopped the script before it was finished.



```
find_hash_with_leading_zeros(base_string, 6)
```

Result:

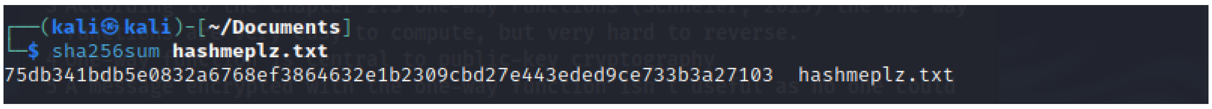


```
┌──(kali㉿kali)-[~/Documents]
└─$ python hash_finder.py
Found! Adding '8464615' to 'Tero' gives a hash starting with 6 zeros: 0000001b3ebb11ca0176226a4c5dcc
b0beec33f89

┌──(kali㉿kali)-[~/Documents]
└─$ echo -n 'Tero8464615'|sha256sum
0000001b3ebb11ca0176226a4c5dcce9fc36f9ebb9e051d7a3f14b0beec33f89  -
```

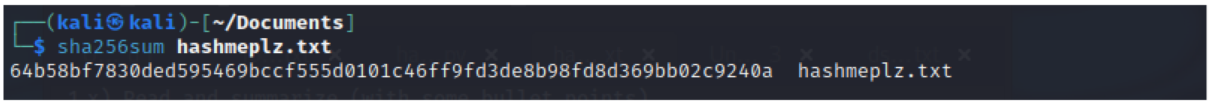
b) Compare hash. Create a small text file. Take it's hash (e.g. 'sha256sum tero.txt'). Change one letter. Take the hash again. Compare hashes. What do you notice?

I copy pasted the above text to a file and took out it's sha256 hash:



```
┌──(kali㉿kali)-[~/Documents]
└─$ sha256sum hashmeplz.txt
75db341bdb5e0832a6768ef3864632e1b2309cbd27e443eded9ce733b3a27103  hashmeplz.txt
```

After one letter was changed the hash changed into this:


```
┌──(kali㉿kali)-[~/Documents]
└─$ sha256sum hashmeplz.txt
64b58bf7830ded595469bccf555d0101c46ff9fd3de8b98fd8d369bb02c9240a  hashmeplz.txt
```

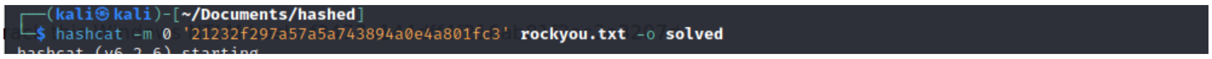
The result is obvious: the hashes are completely different.

b) Install hashcat and test that it works.

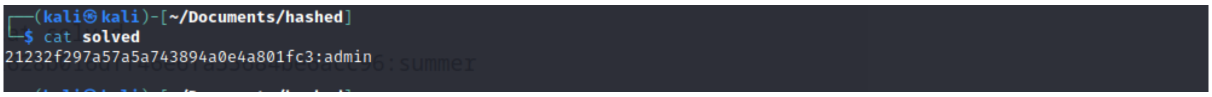
Check

c) Crack this hash: 21232f297a57a5a743894a0e4a801fc3

I used mode 0:


```
┌──(kali㉿kali)-[~/Documents/hashed]
└─$ hashcat -m 0 '21232f297a57a5a743894a0e4a801fc3' rockyou.txt -o solved
```

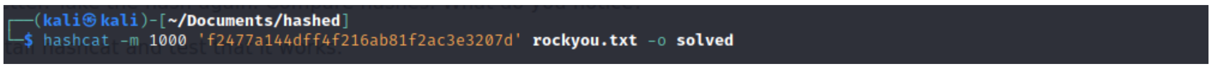
The password is what I feared:

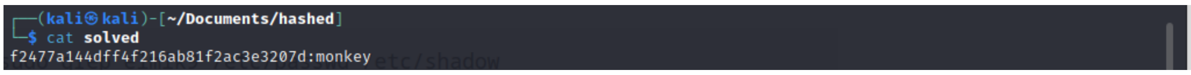

```
┌──(kali㉿kali)-[~/Documents/hashed]
└─$ cat solved
21232f297a57a5a743894a0e4a801fc3:admin
```

'admin'...

d) Crack this Windows NTLM hash: f2477a144dff4f216ab81f2ac3e3207d

I used the hashchat mode 1000 which is meant for NTLM hashes:

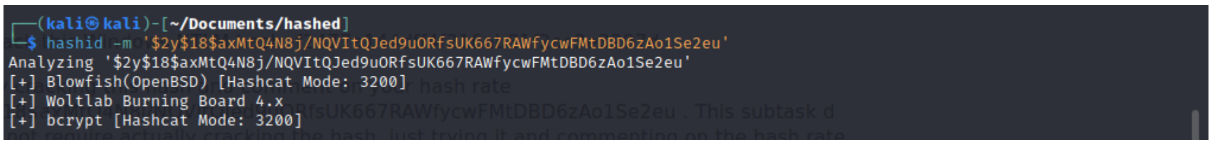

```
┌──(kali㉿kali)-[~/Documents/hashed]
└─$ hashcat -m 1000 'f2477a144dff4f216ab81f2ac3e3207d' rockyou.txt -o solved
```


```
┌──(kali㉿kali)-[~/Documents/hashed]
└─$ cat solved
f2477a144dff4f216ab81f2ac3e3207d:monkey
```

Password is 'monkey'

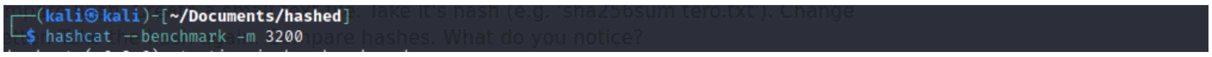
e) Try cracking this hash and comment on your hash rate $2y$18$axMtQ4N8j/NQVltQJed9uORfsUK667RAWfycwFMtDBD6zAo1Se2eu . This subtask d does not require actually cracking the hash, just trying it and commenting on the hash rate.

First let's define the hash type:



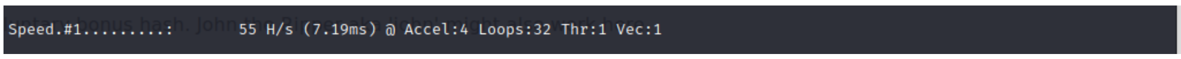
```
┌──(kali㉿kali)-[~/Documents/hashed]
└─$ hashid -m '$2y$18$axMtQ4N8j/NQVItQJed9uORfsUK667RAWfycwFMtDBD6zAo1Se2eu'
Analyzing '$2y$18$axMtQ4N8j/NQVItQJed9uORfsUK667RAWfycwFMtDBD6zAo1Se2eu'
[+] Blowfish(OpenBSD) [Hashcat Mode: 3200]
[+] Woltlab Burning Board 4.x
[+] bcrypt [Hashcat Mode: 3200]
```

The mode is blowfish / 3200 so let’s run hashcat benchmark against it:



```
┌──(kali㉿kali)-[~/Documents/hashed]
└─$ hashcat --benchmark -m 3200
```

The result indicates that the benchmark for blowfish on my laptops Intel Core i5-8250U CPU runs at a speed of 55 hashes per second (H/s):



```
Speed.#1.........:        55 H/s (7.19ms) @ Accel:4 Loops:32 Thr:1 Vec:1
```