

PV204 Security Technologies

1st assignment – Password cracking.

Supervisor: Dr.Petr Švenda

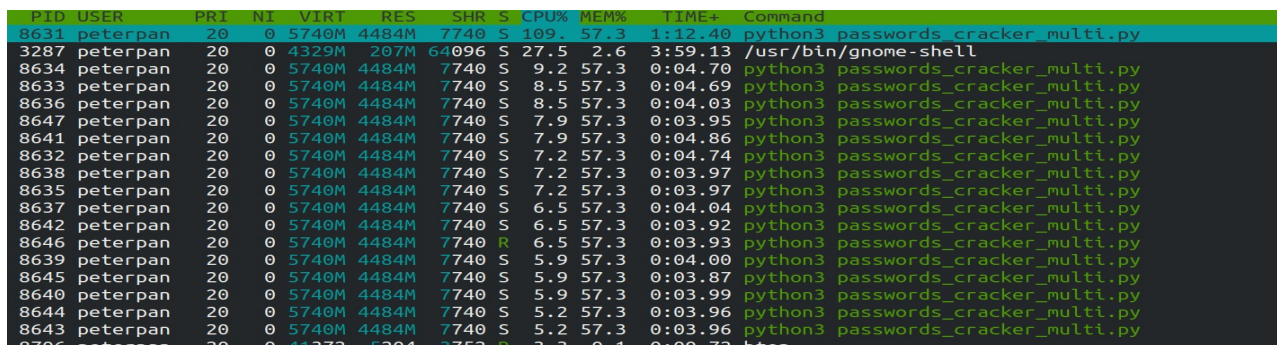
Student: Pedro Gomes, 490830

Due date: 28th February

Description of tooling used

A python script[1] that uses one or more dictionaries to find matches in passwords. Briefly, what it does is that it takes the file with the hashed passwords to crack, detects the different type of hashes on that file, gets the salt out of the hashed password string and hashes words in one or more dictionaries with that salt, trying that way to find a match.

Clearly, this process could take a long time if multi-threading was not used. Because of that I implemented a ThreadPool with a maximum of 16 threads that can run concurrently. The reason why I chose 16, is that my computer **has 4 cores**, each running at 1.6GHz base, with boost up to 3.4GHz, and each core has hyper threading, which virtually **makes 8 cores**. If I run 2 threads per core, that gives me 16 threads that can run concurrently without affecting negatively the overall performance of my machine.



PID	USER	PRI	NI	VIRT	RES	SHR	S	CPU%	MEM%	TIME+	Command
8631	peterpan	20	0	5740M	4484M	7740	S	109.5	57.3	1:12.40	python3 passwords_cracker_multi.py
3287	peterpan	20	0	4329M	207M	64096	S	27.5	2.6	3:59.13	/usr/bin/gnome-shell
8634	peterpan	20	0	5740M	4484M	7740	S	9.2	57.3	0:04.70	python3 passwords_cracker_multi.py
8633	peterpan	20	0	5740M	4484M	7740	S	8.5	57.3	0:04.69	python3 passwords_cracker_multi.py
8636	peterpan	20	0	5740M	4484M	7740	S	8.5	57.3	0:04.03	python3 passwords_cracker_multi.py
8647	peterpan	20	0	5740M	4484M	7740	S	7.9	57.3	0:03.95	python3 passwords_cracker_multi.py
8641	peterpan	20	0	5740M	4484M	7740	S	7.9	57.3	0:04.86	python3 passwords_cracker_multi.py
8632	peterpan	20	0	5740M	4484M	7740	S	7.2	57.3	0:04.74	python3 passwords_cracker_multi.py
8638	peterpan	20	0	5740M	4484M	7740	S	7.2	57.3	0:03.97	python3 passwords_cracker_multi.py
8635	peterpan	20	0	5740M	4484M	7740	S	7.2	57.3	0:03.97	python3 passwords_cracker_multi.py
8637	peterpan	20	0	5740M	4484M	7740	S	6.5	57.3	0:04.04	python3 passwords_cracker_multi.py
8642	peterpan	20	0	5740M	4484M	7740	S	6.5	57.3	0:03.92	python3 passwords_cracker_multi.py
8646	peterpan	20	0	5740M	4484M	7740	R	6.5	57.3	0:03.93	python3 passwords_cracker_multi.py
8639	peterpan	20	0	5740M	4484M	7740	S	5.9	57.3	0:04.00	python3 passwords_cracker_multi.py
8645	peterpan	20	0	5740M	4484M	7740	S	5.9	57.3	0:03.87	python3 passwords_cracker_multi.py
8640	peterpan	20	0	5740M	4484M	7740	S	5.9	57.3	0:03.99	python3 passwords_cracker_multi.py
8644	peterpan	20	0	5740M	4484M	7740	S	5.2	57.3	0:03.96	python3 passwords_cracker_multi.py
8643	peterpan	20	0	5740M	4484M	7740	S	5.2	57.3	0:03.96	python3 passwords_cracker_multi.py
8706	peterpan	20	0	4137M	5204M	3752	R	3.3	0.1	0:00.72	htop

Illustration 1: htop screenshot

If we look at illustration 1, we see that there are 18 threads instances of the “passwords_cracker_multi.py”. One of the threads is the main thread that launched the 16 others of the ThreadPool, the extra 18th one, is perhaps some thread that was about to die in a context switch to another thread that had already entered the ThreadPool. Hence making 1(main thread) + 16(ThreadPool) + 1(Thread about to die) = 18 threads. If multi-thread would not have been used, each password would have need to wait for the computations of the previous one(s). Hence, the final password to be tested against the available dictionaries would need : n seconds to be cracked(successfully or not) + y seconds(all the other previous passwords). Thanks to multi-threading the y seconds variable can be almost ignored, giving us n seconds time per password to be cracked(successfully or not).

Describe characteristics used to generate password hash

According to some online resources[2] the hash algorithms used in */etc/shadow* are the following: MD5, Blowfish, eksBlowfish, SHA-256, SHA-512. After analyzing the meaning of each entry in the file “to_crack.txt” , it was trivial to adapt my script: “passwords_cracker_multi.py” to dissect the information I needed, example:

```
user4:$6$vb1tLY1qiY$Kb90V8ZPJISvGKY2fbTPQgHprj9xAtK6Wak8rPqgfe0HhpgyRULf3m7ydC6pynNQUrVEagWh9kX.CA0G0RiKE.:15405:0:99999:7:::
```

```
user4: username ; $6$: hashing algorithm used(SHA-512) ; $vb1tLY1qiY$ : Salt used
$Kb90V8ZPJISvGKY2fbTPQgHprj9xAtK6Wak8rPqgfe0HhpgyRULf3m7ydC6pynNQUrVEagWh9k
X.CA0G0RiKE. : Hashed Password
```

Curiously, the first three entries in “to_crack” do not have a hashing algorithm ID. This was the perfect situation to use a script[3] I made years ago when reading “Violent Python”, with the

purpose to detect the type of hash of some string. After running it for the first three entries, I got the following results:

```
[>]bee783ee2974595487357e195ef38ca2 – MD5, SSHA-1, NTLM
```

```
[>]21BD12DC183F740EE76F27B78EB39C8AD972A757 – SHA-1(Hex), TIGER-160(HMAC)
```

```
[>]2F2BB917A7B0317ED404511AFA79514A2133DFD8 - SHA-1(Hex), TIGER-160(HMAC)
```

It was then easy to conclude that the hashing algorithms were: MD5, SHA-1 and SHA-1 respectively(Because of the possibilities of hashing algorithms available for /etc/shadow file).

Discussion how to speedup your setup for faster cracking

JohnTheRipper could have been used directly in order to crack the passwords. Not only JohnTheRipper is written in C, but it also has more optimizations then my script that was made in a couple of hours. I simply did not want to give some commands and see the work being done behind the scenes. I wanted to have fun and to have an overall idea how such a setup could be implemented, manually, from the scratch. **Another idea would be to use Hashcat** with the CUDA cores of my NVIDIA 150mx(2GB). After waiting for +/- 7 hours and only being able to successfully crack 1 password with my script(Using the same dictionary of 716MB), **I switched to HashCat**, and I was able to crack 9 more passwords, with different and larger dictionaries, in less time! I must say that doing this attack with the GPU was very much joyful. Please refer to the table below to see the final results:

Table

User	Pwd	Tools	Tools setup	HW platform	Time to crack(sec onds)	#leaked uses	Any notes
user4	PASSWORD	My home made script	python3 passwords _cracker_ multi.py	Ubuntu 18.04.2 LTS 4xcores 1.6GHz – 3.6GHz (each with hyper threading)	193	x56764 times	The dictionarie s needed to be in the cwd of the script. Please refer to the code to see + details
user7	eiz	Hashcat	./ hashcat64. bin -m 500 -a 0 -o found1.txt tocrack1.h ash realhuman _phill.txt -- username	GPU NVIDIA 150mx 2GB	150	x58 times	Where 500 stands for MD5, -- username ignores the <username :...>, found1.txt the output file.
user10	utro	Hashcat	./ hashcat64. bin -m 500 -a 0 -o	GPU NVIDIA 150mx 2GB	260	x117 times	Where 500 stands for MD5, -- username

			found1.txt tocrack1.h ash realhuman _phill.txt -- username				ignores the <username :...>, found1.txt the output file.
user8	k8k	Hashcat	./ hashcat64. bin -m 500 -a 0 -o found1.txt tocrack1.h ash 18_in_1.lst --username	GPU NVIDIA 150mx 2GB	9min:56se c	X20 times	Where 500 stands for MD5, -- username ignores the <username :...>, found1.txt the output file.
user12	k*G9	Hashcat	./ hashcat64. bin -m 500 -a 0 -o found1.txt tocrack1.h ash 18_in_1.lst --username	GPU NVIDIA 150mx 2GB	10min:34s ec	No matches found	Where 500 stands for MD5, -- username ignores the <username :...>, found1.txt the output file.
user11	ro7v	Hashcat	./ hashcat64. bin -m 500 -a 0 -o found1.txt tocrack1.h ash 18_in_1.lst --username	GPU NVIDIA 150mx 2GB	14min:23s ec	X1 time	Where 500 stands for MD5, -- username ignores the <username :...>, found1.txt the output file.
user9	#8R	Hashcat	./ hashcat64. bin -m 500 -a 0 -o found1.txt tocrack1.h ash 18_in_1.lst --username	GPU NVIDIA 150mx 2GB	1h:23min: 21sec	No matches found	Where 500 stands for MD5, -- username ignores the <username :...>, found1.txt the output file.
user13/1 6	kdu5n	Hashcat	./ hashcat64. bin -m 500 -a 0 -o	GPU NVIDIA 150mx 2GB	2h:5min	No matches found	Where 500 stands for MD5, -- username

			found1.txt tocrack1.h ash 18_in_1.lst --username				ignores the <username :...>, found1.txt the output file.
user15/1 8	kF5)d	Hashcat	./ hashcat64. bin -m 500 -a 0 -o found1.txt tocrack1.h ash 18_in_1.lst --username	GPU NVIDIA 150mx 2GB	2h:20min	No matches found	Where 500 stands for MD5, -- username ignores the <username :...>, found1.txt the output file.
user25	miracle77	Hashcat	./ hashcat64. bin -m 7400 -a 0 - o found5.txt tocrack1.h ash 18_in_1.lst --username	GPU NVIDIA 150mx 2GB	3h:22min	X319 times	Where 7400 stands for SHA256 (Unix), -- username ignores the <username :...>, found5.txt the output file.

It would be also possible to use rainbow tables, but I would need to compute them my self, because of the salts used for the different passwords. This process would perhaps take even more time than my home made script.

To sum up, I would like to mention that SHA-256 and SHA-512 took a lot more time to compute than MD5. Clearly, because of the way the algorithms are designed.

Finally, if I need to do this attack again in the future, I will directly use my GPU, and if I design an hashing algorithm one day, I will make sure that even with the GPU it would take a lot of time to make such an attack.

References:

[1] The python script I made to crack the passwords. Please see the attachments in the zip file, and refer to: `passwords_cracker_multi.py` .

[2] <http://www.yourownlinux.com/2015/08/etc-shadow-file-format-in-linux-explained.html> :
Explanations about the syntax of the `/etc/shadow` and `/etc/passwd`

[3] https://github.com/OblackatO/OffensiveSecurity/blob/master/Hashing/Find_hash_type.py :
Script made by me to identify different types of hashing algorithms.

Dictionaries used:

JohnTheRipper default dic at: `/usr/share/john/passwords.lst`:

<https://xato.net/today-i-am-releasing-ten-million-passwords-b6278bbe7495> : 194MB of words
`real_humanphill.txt` : 716MB of words

Several dictionaries available from : <https://thehacktoday.com/password-cracking-dictionaries-download-for-free/> , including: [10-million-combos.zip](#) 85MB, [wordlist by Kakoluk.rar](#) 1.2GB
`18_in_1.lst` : 36GB dictionary