Technische Universität Wien

188.922 Digital Forensics

Lecturers: Dipl.Inf. (FH) Karsten Theiner and DI Dr. Martin Schmiedecker

Teaching Assistant: Christoph Kraus

Assignment 1: Veracrypt

Bruno André Moreira Rosendo

e12302727

Due Date: October 25, 2023

Submitted: October 18, 2023

Contents

1 Purpose

The objective of this assignment was to aid Spongebob in the recovery of a forgotten password for his Veracrypt container through the application of a brute-force approach. This endeavour was part of a Digital Forensics course, where students were tasked with developing a method to regain access to a locked Veracrypt container under certain constraints, primarily focused on password complexity and length.

To initiate the process, the students were provided with a dedicated page that facilitated the generation of a container for Veracrypt. This tool allowed them to specify the desired complexity and length of the password. The complexity could encompass a range of options, which may include elements such as uppercase letters, lowercase letters, numbers, or special characters. Additionally, the length of the password could be customized, depending on the level of security desired.

The primary challenge in this project was the application of a brute-force method to recover the password. Brute force is a technique in digital forensics and cybersecurity where an automated program systematically tries all possible combinations of characters until the correct password is found.

The importance of considering password complexity is paramount in this task. The complexity of a password directly impacts its vulnerability to brute-force attacks. A more complex password, combining different character types and a longer length, makes it significantly harder for an attacker to guess or crack the password. This emphasizes the importance of users choosing strong and complex passwords for their digital security.

2 Findings (Befund)

The initial step of the project involved creating a container with a user-defined password complexity and length. To speed up the cracking process, the decision was made to opt for the simplest complexity (comprising only numbers) and the shortest length (4 characters).

Prior to starting the password-cracking procedure, it is crucial to hash the container to guarantee that any modifications made to it do not alter its content in any manner. To ensure the ability to return to the initial state, one option could be to redownload it; however, a safer approach is to duplicate its contents into a new file. The hashes of both the original and the duplicate were identical:

```
md5sum container_12302727.hc > hash.txt
md5sum container_12302727_copy.hc > hash_copy.txt
cat hash.txt
ae01319340bcdba36f99abb94aa89d52  container_12302727.hc
cat hash_copy.txt
ae01319340bcdba36f99abb94aa89d52  container_12302727_copy.hc
```

Upon obtaining the container and storing its hash, the subsequent action entails extracting the password's hash. To achieve this, we can simply read the initial 512 bits of the file, as detailed by CodeOnBy [1]. We can utilize the *dd* Linux command with the following parameters:

```
dd if=container_12302727.hc of=target_hash.tc bs=512 count=1
```

Here, *if* specifies the filename to be targeted, while *of* designates the file where the bits will be stored. The last two parameters, bs and *count* indicate the retrieval of a single 512-bit block. The actual value is not displayed in this report due to formatting constraints.

After obtaining the password's hash, we should verify its contents were not changed by hashing it again and comparing it to the initial one. We can then employ the *hashcat*

[2] (v6.2.6) program to perform a brute-force attack on all 4-digit passwords consisting exclusively of numbers:

```
hashcat -a 3 target_hash.tc ?d?d?d?d -o result.txt -m 13711
```

The following arguments are relevant:

- ***-a 3*** specifies the cracking method as brute force.
- ***?d?d?d?d*** represents the password mask, where ***?d*** denotes any single digit, and ***?*** is a placeholder for a variable.
- ***-m 13711*** defines the hash mode as *VeraCrypt RIPEMD160 + XTS 512 bit (legacy)* which is based on the problem's original description regarding AES with RIPE-MD 160.
- ***-o result.txt*** specifies the filename for storing the password if the cracking attempt is successful.

After running for approximately 8 minutes (duration may vary depending on the machine), a match for the provided hash was obtained. It's worth noting that in this case, the cracking process was conducted in a CPU within a Kali (v2023.3) virtual machine (VirtualBox v6.1.38) due to driver issues with *hashcat* on the primary system. This machine was capable of about 18 hashes/guesses per second.

Ultimately, a simple ***cat result.txt*** command can be executed to reveal the password! Afterwards, we can use Veracrypt's CLI (v1.24-Update7), or graphical interface, to mount the container:

```
veracrypt -t ./container_12302727
```

This way, the goal was achieved! Note that before accessing the container, we should first ensure its contents were not modified by hashing it again and comparing it to the initial one.

3 Analysis (Gutachten)

The subsequent section responds to all the assignment's specific questions and assesses how password complexity and length contribute to defence against brute-force attacks.

3.1 What is the password of the container?

In this case, the password was 4573. Note that this depends on the generated container and the password complexity chosen before generating it.

3.2 What is the "secret" in the container?

The secret.txt file inside the contained has the following string:

```
7204fb1d9bee6a7a8ec20c135ce4d4da109c570071
```

3.3 What was saved in the container by Spongebob?

Besides the secret.txt file, SpongeBob stored a few pictures inside the container, as seen in figures 3.3, 3.3, and 3.3.

3.4 How much time is needed for brute forcing different password lengths and character sets?

To assess password complexity efficiently, without enduring extended cracking process delays, we'll assume a conservative rate of 2771 guesses per second. This value was taken from an RTX 3090 benchmark published by Chick3nman [3]. In other words, we'll illustrate this using an instance of a machine capable of generating 2771 password hashes per second and estimate the time required to attempt all conceivable passwords (in the worst-case scenario).

Figure 1: awesome.jpg

Figure 2: tripping.jpg

Figure 3: wasted.jpg

This approach involves employing the subsequent formula for these calculations:

$$total\_seconds = \frac{CHAR\_OPTIONS^{PASSWORD\_LENGTH}}{GUESSES\_PER\_SECOND} \quad (1)$$

The parameters *CHAR_OPTIONS*, *PASSWORD_LENGTH*, and *GUESSES_PER_SECOND* have been predetermined as input variables prior to the calculation. This submission includes a Python (v3.10.12) script that automates the computation and formats the results in a user-friendly manner. Additionally, a separate script has been created for generating visual charts, which illustrate the time required for each level of password complexity.

Now, let's explore various password complexities by altering the number of potential characters, such as numbers or letters, and conducting these calculations between the lengths of 4 and 15 characters.

3.4.1 Numbers [0-9]

As previously shown, the numeric character set provides 10 options for each character within the password. In this context, the plot depicting password length versus the time required for cracking appears as shown below:

Table 1: Time to crack different password lengths with only numbers

Password Length	**Time to Crack**	**Password Length**	**Time to Crack**
4	4 seconds	10	41 days 18 hours
5	36 seconds	11	1 year 52 days
6	6 minutes	12	11.5 years
7	1 hour	13	114.5 years
8	10 hours	14	1 144 years
9	4 days 4 hours	15	11 443 years

In this scenario, it's worth noting that all passwords with a length of up to 8 characters can be cracked within a single day. However, for passwords with 11 characters, the time required extends beyond one year, and for those with 15 characters, it escalates to over 11 000 years for a brute-force attack to succeed!

3.4.2 Lower Case [a-z] and Upper Case [A-Z]

When considering exclusively lowercase and uppercase letters, each character offers a pool of 26 potential options:

Table 2: Time to crack different password lengths with upper or lower case characters

Password Length	**Time to Crack**	**Password Length**	**Time to Crack**
4	2 minutes 45 seconds	10	1 615.5 years
5	1 hour 11 minutes	11	42 001 years
6	1 day 7 hours	12	1M years
7	33.5 days	13	28M years
8	2.5 years	14	738M years
9	62 years	15	19B years

In this scenario, a single day would suffice to crack passwords with a length of up to 5 characters, while a password containing 8 characters would require approximately 2.5 years. The challenge becomes progressively formidable with 10-character passwords, where the estimated time extends to more than 1 600 years. Astonishingly, for a password comprising 15 characters, the time needed for a successful brute force attack surpasses an astounding 19 billion years, rendering it virtually unbreakable (that's more time than the age of the Universe!).

3.4.3 Alphabetic Characters [a-zA-Z]

When taking into account both lowercase and uppercase letters, each character provides a selection of 52 possible options:

Table 3: Time to crack different password lengths with alphabetic characters

Password Length	Time to Crack	Password Length	Time to Crack
4	44 minutes	10	1.6M years
5	1 day 14 hours	11	86M years
6	82.5 days	12	4B years
7	11 years 279 days	13	233B years
8	613 years	14	12T years
9	31 812 years	15	629T years

In this situation, a password consisting of 5 characters would already require more than a day to crack, and an 8-character password would require over 600 years. Remarkably, a password with a length of 15 characters would necessitate more than 600 trillion years for a brute-force attack to succeed, highlighting its exceptional security.

3.4.4 Alphanumeric Characters [a-zA-Z0-9]

By incorporating numbers alongside all the letters in the character set, we expand the choices to 62 potential options for each character within the password, as shown in the table below:

This character set yields outcomes similar to the alphabetic set, yet naturally results in even more substantial cracking times. For instance, a password composed of 5 characters would take nearly 4 days to crack, while an 8-character password would demand approximately 2,500 years. Astonishingly, a 15-character password breaches the quadrillion-year

Table 4: Time to crack different password lengths with alphanumeric characters

Password Length	Time to Crack	Password Length	Time to Crack
4	1.5 hours	10	9.6M years
5	3 days 20 hours	11	595M years
6	237 days	12	37B years
7	40 years 109 days	13	2.3T years
8	2499 years	14	142T years
9	154 911 years	15	8.8P years

mark, underlining its formidable level of security.

3.4.5 Special Characters [%&[(=*)+]!#‘;,.]

For the special character set, we'll take into account the 17 options shown in the table above:

Table 5: Time to crack different password lengths with only special characters

Password Length	Time to Crack	Password Length	Time to Crack
4	30 seconds	10	23 years
5	8.5 minutes	11	392 years
6	2 hours 25 minutes	12	6 667 years
7	1 day 17 hours	13	113 342 years
8	29 days	14	1.9M years
9	1 year 130 days	15	32.8M years

In this scenario, as anticipated, the outcome falls between the numeric and alphabetic sets. It's feasible to crack an 8-character password within a month, but it becomes im-

possible from 11 characters onward, where the estimated time reaches nearly 400 years. Notably, a 15-character password would need 32 million years to crack.

3.4.6 All Characters [a-zA-Z0-9%&[(=*)+]!#‘;,.]

The ultimate character set encompasses all the options we've encountered thus far, providing a total of 79 possibilities for each character within the password:

Table 6: Time to crack different password lengths with all characters

Password Length	Time to Crack	Password Length	Time to Crack
4	4 hours	10	108M years
5	12 days 20 hours	11	8.5B years
6	285 days	12	676B years
7	220 years	13	53.4T years
8	17 361 years	14	4.2P years
9	1.4M years	15	333P years

In this scenario, the feasibility of cracking commences to diminish at the 7-character mark, taking over 200 years. It's worth noting that a 5-character password could be breached in just 13 days, whereas a 15-character password extends beyond a staggering 300 quadrillion years!

3.5 What password complexity is needed for a 10-year secure container?

Determining the complexity required for a secure container over a 10-year duration can be easily achieved by reversing the previous formula (as 10 years equate to 315 569 260 seconds):

$$password\_length = \frac{log(315569260 GUESSES\_PER\_SECOND)}{log(CHAR\_OPTIONS)} \quad (2)$$

This computation enables us to identify the recommended password lengths for various password types, ensuring that all of them exceed 10 years in cracking time, as depicted in the table below. The submission also includes a Python (v3.10.12) script for these calculations.

Table 7: Number of characters for passwords safe for 10 years

Password Complexity	**No. of Characters**
Numbers	12
Lower/Upper Case	9
Alphabetic	7
Alphanumeric	7
Special Characters	10
All Characters	7

4 Literature Cited

References

[1] CodeOnBy. Brute-force veracrypt encryption, 2022. URL https://codeonby.com/2022/01/19/brute-force-veracrypt-encryption/.

[2] Tero Karvinen. Cracking passwords with hashcat, 2022. URL https://terokarvinen.com/2022/cracking-passwords-with-hashcat/.

[3] Chick3nman. Hashcat v6.1.1 benchmark on the nvidia rtx 3090, 2020. URL https://gist.github.com/Chick3nman/e4fcee00cb6d82874dace72106d73fef.