

# **Anti-Digital Forensics**

**Cracking and Analysing Encryption Technologies using  
Hashcat**



**Advanced Password  
Recovery**

**Realised by :** Yanis Alim  
Anastasia Cotorobai

**Reported to :** Félix Pichard



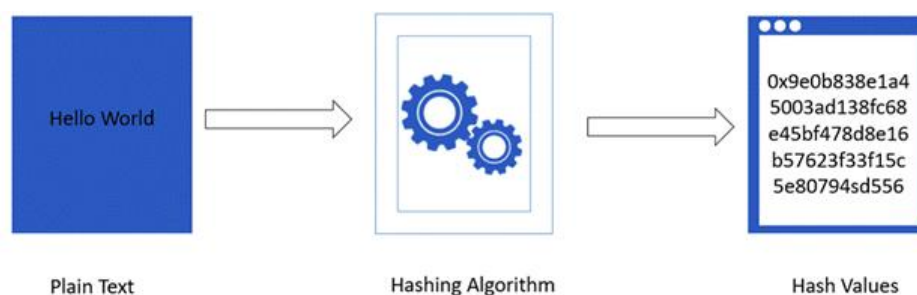
# Table of contents

<b>Introduction</b>	<b>3</b>
<b>BitLocker</b>	<b>4</b>
Description	4
Encryption/Decryption Procedure	5
Disk Encryption	6
Hash Extraction	8
Dictionary Attack	9
Mask Attack	11
Complexity analysis	12
Conclusion	17
<b>Luks</b>	<b>18</b>
The Setup	19
Luks - version 2	21
Bruteforce-luks	22
Dictionary attack	22
Mask attack	23
Complexity analysis	25
Conclusion	29

# Introduction

Today, at a time when the world of computing is booming, the features provided by increasingly sophisticated data protection has become a major issue. Each of us has already registered on a site providing, therefore, personal information, now access to his online bank account or can access his internet customer account in order to to have access to the details of its consumption, to the new offers available...

Obviously, an authentication step via a password and a username (or an email address) is necessary allowing the user to connect and prove their identity. It is clear that this data is stored somewhere on the server. The question of security is therefore essential: anyone who has access to this information can then usurp your identity and thus do any sort of thing in your name (purchases, retrieving bank data, etc.). passwords are never stored in the clear in the database but in the form of a hash of fixed size calculated using a hash function :



The functions have several characteristics:

- it must be computationally impossible to find the clear from the hash
- it must be impossible to find two clears giving the same hash

There are several types of hash functions: MD5, SHA, Bcrypt, AES which offers more or less good performance. The fact that passwords are not stored increases security, however very powerful software has been developed to find the clear from the fingerprint: these software are password breakers. The best known are **Hashcat** and **bruteforce-luks** who claim to be the fastest in the world. Note that these software are designed to test the security of passwords and not hacking.

How to secure your password? First of all the hash function that calculates the fingerprint is very important: we know MD5 is already obsolete.

But what about technologies that are mostly used nowadays ? Are they really hard to break ? Let's take a look at **BitLocker** & **LUKS**.

# BitLocker

## Description

BitLocker is a data protection feature that integrates with the Windows operating system and addresses the threats of data theft or exposure from lost, stolen, or inappropriately decommissioned computers. It offers a number of different authentication methods, like Trusted Platform Module, Smart Key, Recovery Password, user supplied password.

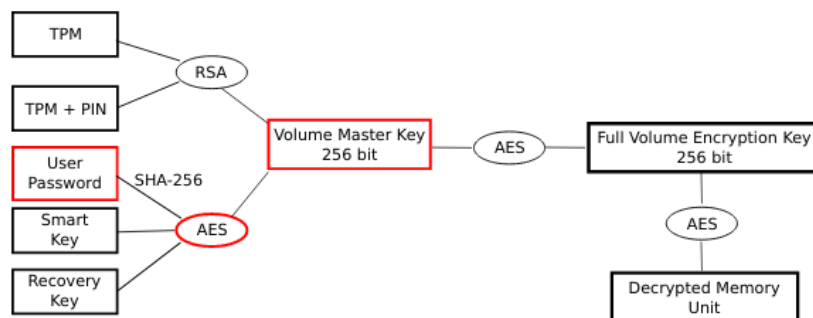
BitLocker features a pretty complex proprietary architecture but it also leverages some well-known algorithms, like **SHA-256** and **AES**. It is possible, and relatively easy to instantly decrypt disks and volumes protected with BitLocker by using the decryption key extracted from the main memory (RAM). In addition, it is also possible to decrypt for offline analysis or instantly mount BitLocker volumes by utilizing the escrow key (BitLocker Recovery Key) extracted from a user's Microsoft Account or retrieved from ActiveDirectory.

## Encryption/Decryption Procedure

During the encryption procedure, each sector in the volume is encrypted individually, with a part of the encryption key being derived from the sector number itself. This means that two sectors containing identical unencrypted data will result in different encrypted bytes being written to the disk, making it much harder to attempt to discover keys by creating and encrypting known data. BitLocker uses a complex hierarchy of keys to encrypt devices. The Sectors themselves are encrypted by using a key called the Full-Volume Encryption Key (FVEK).

To gain an insight about the workings of our attack, more information is necessary about the VMK decryption procedure when the authentication method is a user password:

1. The user provides the password
2. SHA-256 is executed twice on it :



3. There is a loop of 0x100000 iterations, in which SHA-256 is applied to a structure like:

---

```
typedef struct {
    unsigned char updateHash[32]; //last SHA-256 hash calculated
    unsigned char passwordHash[32]; //hash from step 2
    unsigned char salt[16];
    uint64_t hash_count; // number of hash in loop, incremented
                        // at the end of every iteration
} bitlockerMessage;
```

---

4. This loop produces an intermediate key, used with AES to encrypt the Initialization Vector (IV) (derived from a nonce);
5. XOR between encrypted IV and encrypted Message Authentication Code (MAC) to obtain the decrypted MAC;
6. XOR between encrypted IV and encrypted VMK to obtain the decrypted VMK;
7. If the MAC, calculated on the decrypted VMK, is equal to the decrypted MAC, the input password and the decrypted VMK are correct;

All the elements required by the decryption procedure (like VMK, MAC, IV, etc..) can be found inside the encrypted volume. In fact during the encryption, BitLocker stores not only encrypted data but also metadata that provide information about encryption type, keys position, OS version, file system version and so on.

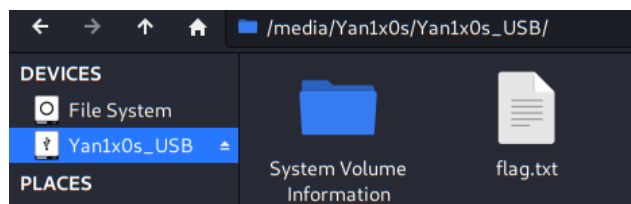
[\[https://arxiv.org/pdf/1901.01337.pdf\]](https://arxiv.org/pdf/1901.01337.pdf)

## Disk Encryption

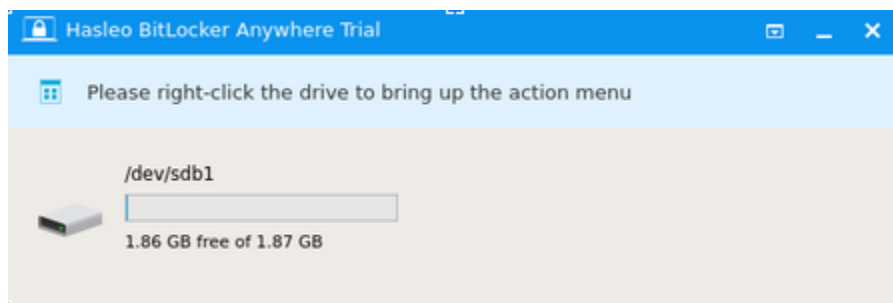
We will be using Kali linux for both :

- Setup an encrypted disk with **BitLocker**
- Recover the password of disk using **Hashcat**

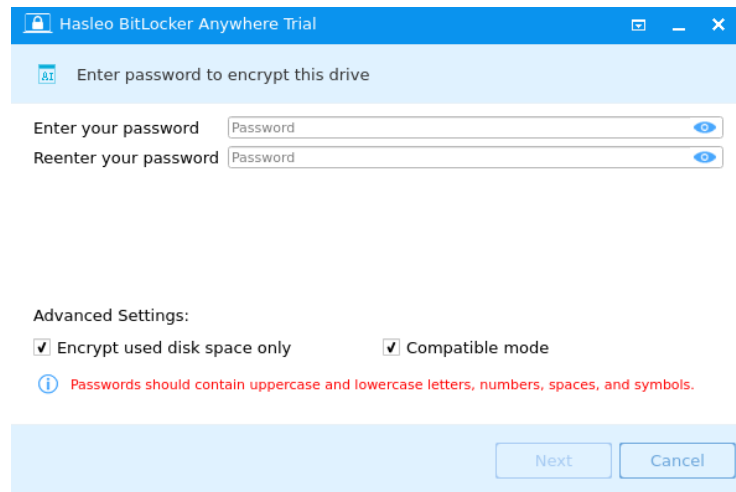
We place a USB Stick :



We launch BitLocker for linux :

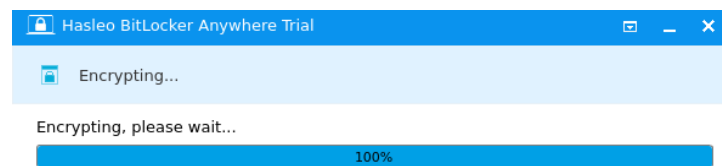


Right click and choose to encrypt disk :



The screenshot shows the 'Hasleo BitLocker Anywhere Trial' application window. The title bar is blue with a lock icon and the text 'Hasleo BitLocker Anywhere Trial'. The main area has a light blue header with a lock icon and the text 'Enter password to encrypt this drive'. Below this, there are two password input fields: 'Enter your password' and 'Reenter your password', each with a 'Password' placeholder and a toggle icon. Underneath the fields is the 'Advanced Settings' section, which includes two checked options: 'Encrypt used disk space only' and 'Compatible mode'. A red information icon is followed by the text 'Passwords should contain uppercase and lowercase letters, numbers, spaces, and symbols.' At the bottom right, there are 'Next' and 'Cancel' buttons.

We choose a password and launch the encryption :



The screenshot shows the 'Hasleo BitLocker Anywhere Trial' application window during the encryption process. The title bar is blue with a lock icon and the text 'Hasleo BitLocker Anywhere Trial'. The main area has a light blue header with a lock icon and the text 'Encrypting...'. Below this, the text 'Encrypting, please wait...' is displayed above a blue progress bar that is filled to 100%.

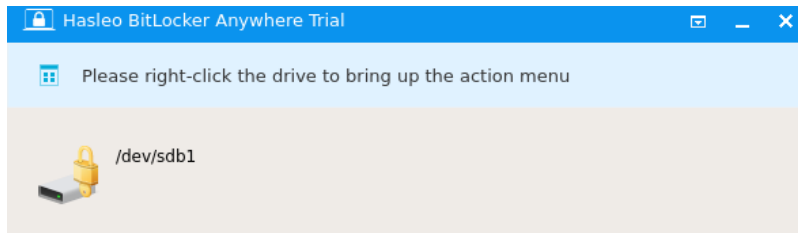
During the encryption of a memory device, (regardless the authentication method) BitLocker asks the user to store somewhere a Recovery Password that can be used to restore the access to the encrypted memory unit in the event that she/he can't unlock the drive normally. Thus the Recovery Password is a common factor for all the authentication methods and it consists of a 48-digit key like this:

```
[Yan1x0s@1337]-[/root/Desktop]
$ls
BitLocker Recovery Key 2597632D-2A23-9140-BCEB-AD94D8B85075.TXT  BitLocker Recovery Key 85F20861-ABDA-044C-B6C5-1E42CB616C17.TXT
[Yan1x0s@1337]-[/root/Desktop]
$ sudo cat BitLocker\ Recovery\ Key\ 2597632D-2A23-9140-BCEB-AD94D8B85075.TXT
00BitLocker Drive Encryption recovery key

To verify that this is the correct recovery key, compare the start of the following identifier with the identifier value displayed on your PC.
Identifier:
    2597632D-2A23-9140-BCEB-AD94D8B85075

If the above identifier matches the one displayed by your PC, then use the following key to unlock your drive.
Recovery Key:
    687434-533874-158587-417626-632368-655468-686037-421685
```

And finally our USB drive is now encrypted :



## Hash Extraction

In order to start the attack, we need to extract the image of your memory device encrypted with BitLocker.

First, let's find the disk location :

```
[Yan1x0s@1337]-[/root/Desktop]
$ sudo lsblk
NAME MAJ:MIN RM  SIZE RO TYPE MOUNTPOINT
sda   8:0    0 238.5G  0 disk
├─sda1 8:1    0   9.3G  0 part [SWAP]
└─sda2 8:2    0 229.2G  0 part /
sdb   8:16   1   1.9G  0 disk
└─sdb1 8:17   1   1.9G  0 part
```

Then, we create a raw copy of the disk :

```
[x]-[Yan1x0s@1337]-[/root/Desktop]
$ sudo dd if=/dev/sdb1 of=/tmp/usb.dd conv=noerror,sync
3932128+0 records in
3932128+0 records out
2013249536 bytes (2.0 GB, 1.9 GiB) copied, 157.568 s, 12.8 MB/s
[Yan1x0s@1337]-[/root/Desktop]
$ _
```

In order to use the BitLocker-OpenCL format, we must produce a well-formatted hash of our encrypted image. We will use the *bitlocker2john* tool (john repo) to extract the hash from the password protected BitLocker encrypted volumes :

```
[x]-[Yan1x0s@1337]-[/tmp]
$ bitlocker2john -i usb.dd
Encrypted device usb.dd opened, size 1919MB

Signature found at 0x3
Version: 8
Invalid version, looking for a signature with valid version...
```

The tool will keep looking for VMK entries and recovering hashes :

```
User Password hash:
$bitlocker$0$16$815a1bb62bba483c88dca07d3db8effa$1048576$12$802db77822d2d60103000000$60$755c
54522b1133be860963cdeabb2e4090dc08a300095da2476141add8a309efbe298835d3150151d1f12d3474643629
```



```
138d9ce18998188fae611de3
```

Hash **type**: User Password with MAC verification (slower solution, no **false** positives)

```
$bitlocker$1$16$815a1bb62bba483c88dca07d3db8effa$1048576$12$802db77822d2d60103000000$60$755c54522b1133be860963cdeabb2e4090dc08a300095da2476141add8a309efbe298835d3150151d1f12d3474643629138d9ce18998188fae611de3
```

Hash **type**: Recovery Password fast attack

```
$bitlocker$2$16$23e4260170fd4bb0a0cae0bc122e9ad2$1048576$12$802db77822d2d60106000000$60$8ad276ea0e022162ede7c918485dfe8c4835eff266c276c328ad47c9d64c4773fdb02fa3dd4da23b75f7c80278ebed0033c41cd5acd6a2189f8a5179
```

Hash **type**: Recovery Password with MAC verification (slower solution, no **false** positives)

```
$bitlocker$3$16$23e4260170fd4bb0a0cae0bc122e9ad2$1048576$12$802db77822d2d60106000000$60$8ad276ea0e022162ede7c918485dfe8c4835eff266c276c328ad47c9d64c4773fdb02fa3dd4da23b75f7c80278ebed0033c41cd5acd6a2189f8a5179
```

It returns 4 output hashes with different prefix:

- If the device was encrypted using the **User Password authentication** method, then we get this 2 hashes :
  - \$bitlocker\$0\$... : it starts the User Password fast attack mode (see [User Password Section](#))
  - \$bitlocker\$1\$... : it starts the User Password attack mode with MAC verification (slower execution, no false positives)
- Else we get this 2 hashes:
  - \$bitlocker\$2\$... : it starts the Recovery Password fast attack mode (see [Recovery Password Section](#))
  - \$bitlocker\$3\$... : it starts the Recovery Password attack mode with MAC verification (slower execution, no false positives)

We will use the first hash because it's **faster**, but we might hit some false positive.

```
[Yan1x0s@1337]—[/tmp]
└─ $echo
'$bitlocker$0$16$815a1bb62bba483c88dca07d3db8effa$1048576$12$802db77822d2d60103000000$60$755c54522b1133be860963cdeabb2e4090dc08a300095da2476141add8a309efbe298835d3150151d1f12d3474643629138d9ce18998188fae611de3' > hash0
```

## Dictionary Attack

Now we will perform dictionary attack, which consists of using a file (wordlist) that contains password candidates, hashcat will apply the same hashing algorithm to each one and compare the result the target hash :

**Options :**

Hashcat can perform multiple types of attacks:

- **Dictionary** (-a 0) – Reads from a text file and uses each line as a password candidate
- **Combination** (-a 1) – Like the Dictionary attack except it uses two dictionaries. Each word of a dictionary is appended to each word in a dictionary.
- **Mask** (-a 3) – Try all combinations in a given key space. It is effectively a brute-force on user specified character sets.
- **Hybrid** (-a 6 and -a 7) – A combination of a dictionary attack and a mask attack.
- Additionally, hashcat also can utilize rule files, which greatly increases the effectiveness of the attack. Hashcat comes with multiple rules, and you can write your own rules as well.

The ‘-m’ indicates the type of hash. Hashcat supports lots of hash types. You can view them using the option --hashes-example, let’s find the mode for BitLocker :

```
[Yan1x0s@1337]-[/tmp]
└─ $hashcat --example-hashes | egrep "*bitlocker" -B 2
MODE: 22100
TYPE: BitLocker
HASH:
$bitlocker$1$16$6f972989ddc209f1eccf07313a7266a2$1048576$12$3a33a8eaff5e6f81d907b591$60$316b
0f6d4cb445fb056f0e3e0633c413526ff4481bbf588917b70a4e8f8075f5ceb45958a800b42cb7ff9b7f5e17c614
5bf8561ea86f52d3592059fb
```

We start our attack using the right parameters :

```
[Yan1x0s@1337]-[/tmp]
└─ $hashcat -m 22100 hash0 -a 0 /usr/share/wordlists/SecLists/Passwords/Leaked-Databases/000webhost.txt --force
hashcat (v6.1.1) starting...

You have enabled --force to bypass dangerous warnings and errors!
This can hide serious problems and should only be done when debugging.
Do not report hashcat issues encountered when using --force.
OpenCL API (OpenCL 1.2 pocl 1.5, None+Asserts, LLVM 9.0.1, RELOC, SLEEF, DISTRO, POCL_DEBUG) - Platform #1 [The pocl project]
=====
* Device #1: pthread-Intel(R) Core(TM) i5-6300U CPU @ 2.40GHz, 2444/2508 MB (1024 MB allocatable), 4MCU

Minimum password length supported by kernel: 4
Maximum password length supported by kernel: 256

Hashes: 1 digests; 1 unique digests, 1 unique salts
Bitmaps: 16 bits, 65536 entries, 0x0000ffff mask, 262144 bytes, 5/13 rotates
Rules: 1

Applicable optimizers applied:
* Single-Hash
* Single-Salt
* Slow-Hash-SIMD-LOOP

Watchdog: Hardware monitoring interface not found on your system.
Watchdog: Temperature abort trigger disabled.

Host memory required for this attack: 65 MB

Initialized device kernels and memory...
```

After 1mn35sec :

```

$bitlocker$0$16$815a1bb62bba483c88dca07d3db8ffa$1048576$12$802db77822d2d60103000000$60$755c54522b1133be860963cdeabb2e4090dc08a300095da2476141add8a309efbe298835d3150151
d1f12d3474643629138d9ce18998188fae611de3[Password@123]
Session.....: hashcat
Status.....: Cracked
Hash.Name.....: BitLocker
Hash.Target.....: $bitlocker$0$16$815a1bb62bba483c88dca07d3db8ffa$10...611de3
Time.Started.....: Mon Dec 14 16:33:49 2020, (1 mln, 35 secs)
Time.Estimated.....: Mon Dec 14 16:35:24 2020, (0 secs)
Guess.Base.....: File (/usr/share/wordlists/SecLists/Passwords/Leaked-Databases/000webhost.txt)
Guess.Queue.....: 1/1 (100.00%)
Speed.#1.....: 21 M/s (5.85ms) @ Accel:8 Loops:4096 Thr:1 Vec:4
Recovered.....: 1/1 (100.00%) Digests
Progress.....: 2023/720304 (0.28%)
Rejected.....: 7/2023 (0.35%)
Restore.Point.....: 1991/720304 (0.28%)
Restore.Sub.#1.....: Salt:0 Amplifier:0-1 Iteration:1044480-1048576
Candidates.#1.....: r3publik -> a1805240

Started: Mon Dec 14 16:33:30 2020
Stopped: Mon Dec 14 16:35:25 2020
[yan1x@s01337]-[/tmp]
$ _

```

Hashcat recovered the password in 1mn35 but we will talk in detail about the complexity of this algorithm later on the analysis section.

## Mask Attack

Mask attacks are similar to brute-force attacks given they try all combinations from a set of characters. With brute-force attacks, all possible characters that exist are tried. Mask attacks are more specific as the set of characters you try is reduced based on information you know.

Hashcat uses regex like option to refer the charset :

- ?l = abcdefghijklmnopqrstuvwxyz
- ?u = ABCDEFGHIJKLMNOPQRSTUVWXYZ
- ?d = 0123456789
- ?h = 0123456789abcdef
- ?H = 0123456789ABCDEF
- ?s = «space»!"#\$%&'()\*+,-./:;<=>?@[\\]^\_`{|}~
- ?a = ?l?u?d?s
- ?b = 0x00 - 0xff

For example, in our situation we recovered the USB from a guy that hides something inside it and gave it to another and gave him a note :

- The password for this USB : Starts with “Password”
- It has the length of 12
- Ends with 3 secret numbers

$\text{len}(\text{“Password”}) = 8 + \text{len}(\text{“3 secret numbers”}) = 11$

This leaves us with 1 character to find between the “Password” and 3 last digits.

We could use this information to create our mask :

- **Password?a?d?d?d**

The mode for this attack is 3 in Hashcat.

```
[Yan1x0s@1337]-[/tmp]
$hashcat -m 22100 hash0 -a 3 Password?a??d?d?d --force
hashcat (v6.1.1) starting...
```

After 3 seconds :

```
$bitlocker$0$16$815a1bb62bba483c88dca07d3db8effa51048576$12$802db77822d2d60103000000$60$755c54522b1133be860963cdeabb2e4090dc08a300095da2476141add8a309efbe298835d3150151d1f12d3474643629138d9ce18998188fae611de3 Password@123

Session.....: hashcat
Status.....: Cracked
Hash.Name.....: BitLocker
Hash.Target.....: $bitlocker$0$16$815a1bb62bba483c88dca07d3db8effa510...611de3
Time.Started.....: Mon Dec 14 17:22:28 2020, (3 secs)
Time.Estimated.....: Mon Dec 14 17:22:31 2020, (0 secs)
Guess.Mask.....: Password?a?d?d?d [12]
Guess.Queue.....: 1/1 (100.00%)
Speed.#1.....: 21 H/s (11.96ms) @ Accel:16 Loops:4096 Thr:1 Vec:8
Recovered.....: 1/1 (100.00%) Digests
Progress.....: 64/95000 (0.07%)
Rejected.....: 0/64 (0.00%)
Restore.Point.....: 0/95000 (0.00%)
Restore.Sub.#1...: Salt:0 Amplifier:0-1 Iteration:1044480-1048576
Candidates.#1...: Passworde123 -> PasswordC123

Started: Mon Dec 14 17:22:22 2020
Stopped: Mon Dec 14 17:22:33 2020
[Yan1x0s@1337]-[/tmp]
$
```

Hashcat recovered the password in 3seconds but we will talk in detail about the complexity of this algorithm later on in the analysis section.

## Complexity analysis

Let's analyze the complexity of the algorithm. From BitLocker2John tool we got this output :

```
[X]-[Yan1x0s@1337]-[/tmp]
$bitlocker2john -i usb.dd
Encrypted device usb.dd opened, size 1919MB

Signature found at 0x3
Version: 8
Invalid version, looking for a signature with valid version...

Signature found at 0x58b000
Version: 2 (Windows 7 or later)

VMK entry found at 0x58b0cf

VMK encrypted with User Password found at 58b0f0
VMK encrypted with AES-CCM
UP Nonce: 802db77822d2d60103000000
UP MAC: 755c54522b1133be860963cdeabb2e40
UP VMK:
90dc08a300095da2476141add8a309efbe298835d3150151d1f12d3474643629138d9ce18998188fae611de3
```

The resulting hash is :

```
$bitlocker$0$16$815a1bb62bba483c88dca07d3db8effa$1048576$12$802db77822d2d60103000000$60$755c54522b1133be860963cdeabb2e4090dc08a300095da2476141add8a309efbe298835d3150151d1f12d3474643629138d9ce18998188fae611de3
```

Let's look at the hash part by part : [<https://arxiv.org/pdf/1901.01337.pdf>]

- **\$bitlocker**: refers the technology
- **\$0**: refers the hash number, since we have 4 hashes as mentioned previously
- **\$16**: size of the next part
- **\$815a1bb62bba483c88dca07d3db8effa** : 16 bytes of salt
- **\$1048576** : iteration number (0x100000) of SHA-256 for the bitlocker Message structure
- **\$12**: size of the next part
- **\$802db77822d2d60103000000** : 12 bytes of Nonce
- **\$60** : size of the next part
- **\$755c54522b1133be860963cdeabb2e4090dc08a300095da2476141add8a309efbe298835d3150151d1f12d3474643629138d9ce18998188fae611de3** : 60 bytes of data (MAC+VMK)
  - **755c54522b1133be860963cdeabb2e40** : 16 bytes MAC
  - **90dc08a300095da2476141add8a309efbe298835d3150151d1f12d3474643629138d9ce18998188fae611de3** : 44 bytes of VMK (AES-256 encrypted with "PASSWORD")

The algorithm is robust as shown on the Encryption/Decryption Procedure part.

For each password candidate, Hashcat should do :

**PBKDF(sha256(sha256(UTF16(password)))) with 0x100000 pbkdf rounds.**

### Password complexity:

The password we used is : Password@123

- In term of complexity, it has Uppercase, Lowercase, Digits, Special Character :

Test Your Password		Minimum Requirements	
Password:	<input type="text" value="Password@123"/>	<ul style="list-style-type: none"> <li>Minimum 8 characters in length</li> <li>Contains 3/4 of the following items: <ul style="list-style-type: none"> <li>Uppercase Letters</li> <li>Lowercase Letters</li> <li>Numbers</li> <li>Symbols</li> </ul> </li> </ul>	
Hide:	<input type="checkbox"/>		
Score:	<div><div>93%</div></div>		
Complexity:	Very Strong		

Additions	Type	Rate	Count	Bonus
Number of Characters	Flat	$+(n^4)$	<input type="text" value="12"/>	+ 48
Uppercase Letters	Cond/Incr	$+(len-n)*2$	<input type="text" value="1"/>	+ 22
Lowercase Letters	Cond/Incr	$+(len-n)*2$	<input type="text" value="7"/>	+ 10
Numbers	Cond	$+(n^4)$	<input type="text" value="3"/>	+ 12
Symbols	Flat	$+(n^6)$	<input type="text" value="1"/>	+ 6
Middle Numbers or Symbols	Flat	$+(n^2)$	<input type="text" value="3"/>	+ 6
Requirements	Flat	$+(n^2)$	<input type="text" value="5"/>	+ 10

- In term of entropy :

**Length:** 12  
**Strength:** Reasonable - This password is fairly secure cryptographically and skilled hackers may need some good computing power to crack it. (Depends greatly on implementation!)  
**Entropy:** 52.7 bits  
**Charset Size:** 72 characters

- In term of security, it's not that secure :

## How secure is your password?

**Tip:** When adding a capital or digit to your password, don't simply put the capital at the start and the digit at the end

Show password: ☒

# Password@123

Very Weak

12 characters containing: Lower case Upper case Numbers Symbols

Time to crack your password:

## 0.01 seconds

Review: Oh dear, using that password is like leaving your front door wide open. Your password is very weak because it contains a common password, a dictionary word and a sequence of characters.

Your passwords are never stored. Even if they were, we have no idea who you are!

This password is available inside common wordlists :

```
[Yan1x0s@1337]-[/usr/share/wordlists/SecLists/Passwords]
$egrep -Rw "Password@123"
Leaked-Databases/000webhost.txt:Password@123
HoneyPot-Captures/multiplesources-passwords-fabian-fingerle.de.txt:Password@123
[Yan1x0s@1337]-[/usr/share/wordlists/SecLists/Passwords]
$
```

## Dictionary Attack Complexity :

- Non successful attack : X time waste [ X is the time spent to perform the attack ]

- Successful attack :
  - With our poor setup ( NO GPU):

```
Device #1: pthread-Intel(R) Core(TM) i5-6300U CPU @ 2.40GHz, 2444/2508 MB (1024 MB
allocatable), 4MCU
Speed.#1.....:      21 H/s (4.62ms) @ Accel:4 Loops:4096 Thr:1 Vec:8
```

Let's find the position of our Password inside the wordlist :

```
[Yan1x0s@1337]—[/tmp]
└─$grep -n "Password@123"
/usr/share/wordlists/SecLists/Passwords/Leaked-Databases/000webhost.txt
2000:Password@123
```

Theoretically , we have 21 hash/sec and we need to try 2000 passwords before we reach the good one.

Which means it will take **2000/21 => 95sec => 1mn35sec**

This corresponds exactly to the time spent to find the password in practice, as shown above !

What is the worst case of a successful dictionary attack using this wordlist ?

```
[Yan1x0s@1337]—[/tmp]
└─$wc -l /usr/share/wordlists/SecLists/Passwords/Leaked-Databases/000webhost.txt
720304 /usr/share/wordlists/SecLists/Passwords/Leaked-Databases/000webhost.txt
```

The worst case is that our password is the last one :

**720304/21 = 09:31:40.**

This is not so bad but we can't say that the cracking speed is good because we are limited to a relatively small number of passwords, let's judge the speed by looking at the mask attack.

### Mask Attack Complexity :

- Non successful attack : X time waste [ X is the time spent to perform the attack ]
- Successful attack :
  - With our poor setup ( NO GPU):

```
Device #1: pthread-Intel(R) Core(TM) i5-6300U CPU @ 2.40GHz, 2444/2508 MB (1024 MB
allocatable), 4MCU
Speed.#1.....:      21 H/s (4.62ms) @ Accel:4 Loops:4096 Thr:1 Vec:8
```

The more information you have about the password the more you optimize time.

Let's start with the worst case :

- Knowing nothing about the password : it's all about luck, to maximize the chances, we can make a mask of any charset from 8 to 12 length (--increment option) :

```
[Yan1x0s@1337]—[/tmp]
```

```
└─ $hashcat -m 22100 hash0 -a 3 ?a?a?a?a?a?a?a --increment --force
```

Time.Started.....: Mon Dec 14 19:29:46 2020, (4 secs)

Time.Estimated....: Next Big Bang, (45537438 years, 181 days)

Progress.....: 16/6634204312890625 (0.00%)

Next Big Bang ? This is so long to wait for X"D

- Knowing the length of the password and the suffix :

```
└─ [X]—[Yan1x0s@1337]—[/tmp]
```

```
└─ $hashcat -m 22100 hash0 -a 3 ?a?a?a?a?a?a?a@123 --force
```

Progress.....: 0/81450625 (0.00%)

Time.Started.....: Mon Dec 14 19:26:05 2020, (1 sec)

Time.Estimated....: Fri Mar 12 03:57:11 2021, (87 days, 8 hours)

We went from YEAR to DAYS, but still long to wait !



## Conclusion

BitLocker is relatively strong if the cracker has no one information about the password and tries to crack it using a mask attack for example it will take long even though if he has string GPU/CPU :

Version	GPU	-t	-b	Passwords x kernel	Passwords/sec	Hash/sec
CUDA	GFT	8	13	106.496	303	635 MH/s
CUDA	GTK80	8	14	114.688	370	775 MH/s
CUDA	GFTX	8	24	106.608	933	1.957 MH/s
CUDA	GTP100	1	56	57.344	1.418	2.973 MH/s
CUDA	GTV100	1	80	81.920	3.252	6.820 MH/s
OpenCL	AMDM	32	64	524.288	241	505 MH/s
OpenCL	GFTX	8	24	196.608	884	1.853 MH/s

N.B. Each password requires about 2.097.152 SHA-256

[<https://github.com/e-ago/bitcracker>]

If we take the best hardware performance : 2.973 MHash/sec

The worst case of 8 to 12 password is : 6634204312890625

This would result in :

```
>>> number_passwords = 6634204312890625
>>> hash_per_second = 2.973 * 2 ** 20
>>> str(datetime.timedelta( seconds= number_passwords // hash_per_second ))
'24630 days, 21:35:42'
```

### Fun facts :

When the USB drive is decrypted, we get :

```
[Yan1x0s@1337]—[/media/Yan1x0s/Yan1x0s_USB]
└─ $ls
  flag.txt  'System Volume Information'
[Yan1x0s@1337]—[/media/Yan1x0s/Yan1x0s_USB]
└─ $cat flag.txt
Anti-Forensics{gg_well_played}
```

# Luks

LUKS encryption is widely used in various Linux distributions to protect disks and create encrypted containers. Being a platform-independent, open-source specification, LUKS can be viewed as an exemplary implementation of disk encryption. Offering the choice of multiple encryption algorithms, several modes of encryption and several hash functions to choose from, LUKS is one of the tougher disk encryption systems to break.

LUKS offers users the choice of various encryption algorithms, hash functions and encryption modes, thus providing some 45 possible combinations.

## LUKS encryption algorithms

LUKS supports multiple combinations of encryption algorithms, encryption modes, and hash functions including:

- AES
- Serpent
- Twofish
- CAST-128
- CAST-256.

Being the only hardware-accelerated encryption algorithm, AES is by far the most common of the three.

## LUKS encryption modes

LUKS supports the following encryption modes:

- ECB
- CBC-PLAIN64
- CBC-ESSIV:*hash*
- XTS-PLAIN64

Various Linux distributions may use different default settings when setting up an encrypted volume. For example, Red Hat Linux employs cbc-essiv:sha256 with a 256-bit AES key, which is the default combination for many popular Linux distributions.

## LUKS hash functions

In disk encryption, hash functions are used as part of the Key Derivation Function (KDF). KDF are used to derive the binary encryption key out of the user-provided input (typically, a text-based passphrase). The following hash functions are supported in the LUKS specification:

- SHA-1
- SHA-256
- SHA-512
- RIPEMD160
- WHIRLPOOL

By default, most Linux distros use SHA-256 as a hash function.

## The Setup

We set up a **Debian Buster** desktop VM with full disk encryption in VMWare.

We used the default “Full disk encryption with LVM” option during installation which uses *dm-crypt/LUKS*. The default *cryptsetup* configuration used is *aes-xts-plain64:sha256* which uses a 512 bit encryption key.

We copy Debian's vmdk file(simulates the hard disk of a virtual machine, and stores all digital data of this VM) to **Kali** linux, from where we'll perform the password recovery.

**From host:**

```
scp debian10-disk1.vmdk kali@172.16.141.3:/home/kali/
```

**From kali:**

List vmdk content:

```
kali@kali:~$ 7z l debian10-disk1.vmdk
```

Date	Time	Attr	Size	Compressed	Name
		.....	254803968	254803968	0.img
		.....	5110759424	5110759424	1.img
		.....	1048576	1048576	2
			5366611968	5366611968	3 files

We can see the **0.img** which is the boot partition and **1.img** which is the main, encrypted partition.

Extract the main partition:

```
7z e debian10-disk1.vmdk 1.img
```

```
kali@kali:~$ sudo cryptsetup open 1.img test1
Enter passphrase for 1.img: 
```

Using the luksDump option of cryptsetup command, we extract some informations from luks header:

- Luks version - *2*
- Cipher Algorithm - *aes-xts-plain64*
- Hash function - *sha256*
- The number of hash iterations and, indirectly, the speed of the user's computer - *136107*
- etc

```

kali@kali:~$ sudo cryptsetup luksDump 1.img
[sudo] password for kali:
LUKS header information
Version:      2
Epoch:       3
Metadata area: 16384 [bytes]
Keyslots area: 16744448 [bytes]
UUID:         f2dd0f09-e4db-4647-a3ac-303a5bd290f9
Label:        (no label)
Subsystem:    (no subsystem)
Flags:        (no flags)

Data segments:
 0: crypt
    offset: 16777216 [bytes]
    length: (whole device)
    cipher: aes-xts-plain64
    sector: 512 [bytes]

Keyslots:
 0: luks2
    Key:      512 bits
    Priority: normal
    Cipher:    aes-xts-plain64
    Cipher key: 512 bits
    PBKDF:     argon2i
    Time cost: 4
    Memory:    620600
    Threads:   1
    Salt:      22 76 6c 6a 4f e9 15 f8 08 22 c0 2a 2a 40 a6 bc
                6c ae ef 16 15 77 0d bc 50 77 80 e4 7c 31 57 d0
    AF stripes: 4000
    AF hash:    sha256
    Area offset: 32768 [bytes]
    Area length: 258048 [bytes]
    Digest ID:  0

Tokens:
Digests:
 0: pbkdf2
    Hash:      sha256
    Iterations: 136107
    Salt:      48 c3 53 6b 07 c8 48 79 0d 3c 8d 4f e6 a9 d9 6b
                85 b3 2b 05 16 5e 82 21 6c 21 d9 a5 09 a0 f1 eb
    Digest:    67 d9 c7 25 4e 16 3a f1 da 50 65 4a d2 06 39 cd
                23 74 6c 8d fa 75 f6 7b 25 03 5f 7b 8a c7 d1 38

```

## Luks - version 2

LUKS2 is the second version of the Linux Unified Key Setup for disk encryption management. It is the follow-up of the LUKS1 format that extends capabilities of the on-disk format and removes some known problems and limitations.

Most of the basic concepts of LUKS1 remain in place.

LUKS provides a generic key store on the dedicated area on a disk, with the ability to use multiple passphrases<sup>1</sup> to unlock a stored key.

LUKS2 extends this concept for more flexible ways of storing metadata, redundant information to provide recovery in the case of corruption in a metadata area, and an interface to store externally managed metadata for integration with other tools. While the implementation of LUKS2 is intended to be used with Linux-based dm-crypt disk encryption, it is a generic format.

## Bruteforce-luks

The purpose of this tool is to find the password of a Luks encrypted volume.

It can perform:

- Dictionary attack - try all password in a file
- Mask attack - Try all the passwords given a charset

The program uses the cryptsetup API (C language) directly to test the header keyslot, which effectively implements the cryptsetup --test-passphrase option available from the cryptsetup shell prompt.

The program does not use GPU, relying only on CPU. It can be set up to run on multiple threads.

## Dictionary attack

\*All the metrics taken will present two cases: 4 threads (best case) and 1 thread ( worst case).

```
kali@kali:~/bruteforce-luks$ time ./bruteforce-luks -t 4 -v 10 -f
../pass.txt ../1.img
Warning: using dictionary mode, ignoring options -b, -e, -l, -m and -s.
```

```
Tried passwords: 7
Tried passwords per second: 0.700000
Last tried password: europa
```

```
Tried passwords: 9
Tried passwords per second: 0.692308
Last tried password: europa
```

```
Password found: Twilight789!
```

```
real    0m13.963s
```

```
kali@kali:~/bruteforce-luks$ time ./bruteforce-luks -t 1 -v 10 -f
../pass.txt ../1.img
Warning: using dictionary mode, ignoring options -b, -e, -l, -m and -s.
```

```
Tried passwords: 3
Tried passwords per second: 0.300000
Last tried password: cheryl
```

```
Tried passwords: 6
Tried passwords per second: 0.300000
Last tried password: andres
```

```
Tried passwords: 9
Tried passwords per second: 0.300000
Last tried password: Twilight789!
```

```
Tried passwords: 9
Tried passwords per second: 0.290323
Last tried password: Twilight789!
```

```
Password found: Twilight789!
```

```
real    0m31.595s
```

The dictionary attack takes ~14 seconds on four threads and ~32 seconds on one thread.

## Mask attack

For the mask attack, bruteforce-luks has command line options to specify:

- the minimum password length to try
- the maximum password length to try
- the beginning of the password
- the end of the password
- the character set to use (among the characters of the current locale)

For the mask attack, we tested the following combination: starts with "Twilight", ends with "!", and contains digits "0123456789" in the middle of the password.

```
time ./bruteforce-luks -t 4 -l 12 -v 50 -b "Twilight" -e "!" -s
"0123456789" ../1.img
...
```

```
Tried / Total passwords: 720 / 1000
Tried passwords per second: 0.757895
Last tried password: Twilight723!
Total space searched: 72.000000%
ETA: Tue 15 Dec 2020 03:40:16 AM EST
```

Tried / Total passwords: 755 / 1000  
Tried passwords per second: 0.755000  
Last tried password: Twilight758!  
Total space searched: 75.500000%  
ETA: Tue 15 Dec 2020 03:39:31 AM EST

Tried / Total passwords: 789 / 1000  
Tried passwords per second: 0.757198  
Last tried password: Twilight792!  
Total space searched: 78.900000%  
ETA: Tue 15 Dec 2020 03:38:45 AM EST

Password found: Twilight789!

real **17m26.668s**

```
time ./bruteforce-luks -t 1 -l 12 -v 200 -b "Twilight" -e "!" -s  
"0123456789" ../1.img
```

...

Tried / Total passwords: 672 / 1000  
Tried passwords per second: 0.336000  
Last tried password: Twilight672!  
Total space searched: 67.200000%  
ETA: Tue 15 Dec 2020 04:15:24 AM EST

Tried / Total passwords: 739 / 1000  
Tried passwords per second: 0.335909  
Last tried password: Twilight739!  
Total space searched: 73.900000%  
ETA: Tue 15 Dec 2020 04:12:04 AM EST

Tried / Total passwords: 789 / 1000  
Tried passwords per second: 0.332491  
Last tried password: Twilight789!  
Total space searched: 78.900000%  
ETA: Tue 15 Dec 2020 04:09:42 AM EST

Password found: Twilight789!



real 39m33.295s

This mask attack takes ~17 minutes on four threads and ~39 minutes on one thread.

## Complexity analysis

### Password complexity:

The password we used is : Twilight789!

- In term of complexity, it has Uppercase, Lowercase, Digits, Special Character :

### Password properties

Property	Value	Comment
Password length:	12	OK
Numbers:	3	USED
Letters:	8	USED
Uppercase Letters:	1	USED
Lowercase Letters:	7	USED
Symbols	1	USED
Charset size	94	HIGH (A-Z, a-z, 0-9, symbols)
TOP 10000 password	NO	Password is NOT one of the most frequently used passwords.

- In term of entropy :

**Length:** 12

**Strength:** Reasonable - This password is fairly secure cryptographically and skilled hackers may need some good computing power to crack it. (Depends greatly on implementation!)

**Entropy:** 53.1 bits

**Charset Size:** 72 characters

- In term of security, it's not that secure :

## How secure is your password?

**Tip:** It's often better to have longer passwords than shorter, more complex ones

Show password: ☒

<b>Twilight789!</b>	
Very Weak	
12 characters containing:	✓ Lower case   ✓ Upper case   ✓ Numbers   ✓ Symbols
Time to crack your password: <b>34.97 seconds</b>	Review: Oh dear, using that password is like leaving your front door wide open. Your password is very weak because it contains a common password, a sequence of characters and a dictionary word.

We'd tried 102 passwords and took **0.411290** as the average value of the number of password tries per second when the programs run on one thread and **1.200000**, when the program runs on 4 threads (number of machine processor cores).

### 1 thread:

```
Tried / Total passwords: 101 / 3.22627e+21
Tried passwords per second: 0.412245
Last tried password: 00000000001d
Total space searched: 0.000000%
ETA: 9223372035246804093 s
```

```
Tried / Total passwords: 101 / 3.22627e+21
Tried passwords per second: 0.410569
Last tried password: 00000000001d
Total space searched: 0.000000%
ETA: 9223372035246804093 s
```

```
Tried / Total passwords: 102 / 3.22627e+21
Tried passwords per second: 0.412955
Last tried password: 00000000001e
Total space searched: 0.000000%
ETA: 9223372035246804093 s
```

```
Tried / Total passwords: 102 / 3.22627e+21
Tried passwords per second: 0.411290
Last tried password: 00000000001e
Total space searched: 0.000000%
ETA: 9223372035246804093 s
```

#### 4 threads:

```
Tried passwords: 100
Tried passwords per second: 1.204819
Last tried password: caroline

Tried passwords: 101
Tried passwords per second: 1.202381
Last tried password: amanda

Tried passwords: 102
Tried passwords per second: 1.200000
Last tried password: maverick

Tried passwords: 103
Tried passwords per second: 1.197674
Last tried password: midnight
```

#### Dictionary attack :

We took a dictionary containing 1580 passwords. The right password is at the 1580th position in the list.

```
kali@kali:~$ tail Top1580-probable.txt
crimson
coconut
cheryl
beavis
anonymous
andres
africa
134679
Twilight789!
```

#### 1 thread:

We have **0.411290** passwords/sec and we need to try 1580 passwords before we reach the good one.

$1580 / 0.411290 \approx 3841.57164045 \text{ s} = 64.0261940075 \text{ min}$

**4 threads:**

We have **1.2** passwords/sec and we need to try 1580 passwords before we reach the good one.

$1580 / 1.2 \approx 1316.66666667 \text{ s} = 21.9444444445 \text{ min}$

**Mask Attack Complexity :**

For mask attack we'd shown multiple metrics. The formula used to compute the number of **Combinations = pow(Size, Keyspace)** and the **Time = Combinations / Tries per second**.

**1 thread:**

Keyspace	Size	Tries / second	Combinations	Seconds	Years	Days	Mask
157	12	0.41129	2.70057E+169	6.5661E+169	2.08209658686154E+162 years	7.59965254204461E+164 days	Nothing known
10	3	0.41129	59049	143570.2303	0 years	1 days	Twilight?d?d?d!
33	1	0.41129	1	2.431374456	0 years	0 days	Twilight789?a
26	7	0.41129	9.38748E+21	2.28245E+22	723759509733984 years	264172221052904000 days	T?u?u?u?u?u?u?u789!

**4 threads:**

Keyspace	Size	Tries / second	Combinations	Seconds	Years	Days	Mask
157	12	1.2	2.70057E+169	2.25048E+169	7.13621254341901E+161 years	2.60471757834794E+164 days	Nothing known
10	3	1.2	59049	49207.5	0 years	0 days	Twilight?d?d?d!
33	1	1.2	1	0.833333333333	0 years	0 days	Twilight789?a
26	7	1.2	9.38748E+21	7.8229E+21	248062540632075 years	90542827330707500 days	T?u?u?u?u?u?u?u789!

## Conclusion

Our tests showed that the multithreaded program “bruteforce-luks” is extremely slow. If we take the mask attack, the only way to have a reasonable time is to know a lot of information about the password, in our case the start “Twilight” and the end “!”, which is rarely the case in real life.

We'd also seen that the recovery speed depends on several things, the most important being the following:

- **The password complexity**
- **The hardware:** Most cracking algorithms use the video card, which is not the case for the tool we used to perform luks cracking, making it very slow. Same goes for the number of computers doing the attack; the more computers are crunching the password, the faster the speed will be.

- **The hash function.** Some hash functions are slower than others. In the case of LUKS, the user has the choice of five hash functions including RIPEMD160, SHA-1, SHA-256, SHA-512, and WHIRLPOOL, RIPEMD160 being the fastest and WHIRLPOOL the slowest of the pool. Depending on the choice of the hash function made at the time of setting up the encryption, your attacks may go faster or slower.
- **The encryption algorithm and encryption mode.** Once again, these settings are selected at the time the user sets up the encryption. The choice affects the speed of the attack, but to a significantly minor degree compared to the choice of a hash function.