

# Passwortwiederherstellung

a.k.a. Password-Cracking

---

**Dozent:** Prof. Dr. Michael Eichberg  
**Kontakt:** [michael.eichberg@dhbw-mannheim.de](mailto:michael.eichberg@dhbw-mannheim.de)  
**Version:** 2024-02-28



---

1

- Wer hat schon einmal Passworte wiederhergestellt?
- Wer hat Erfahrung mit Linux?
- Wer hat Erfahrung mit Linux Kommandozeilenwerkzeugen für die Textverarbeitung?
- Wer hat Erfahrung mit regulären Ausdrücken?
- Wer hat Erfahrung mit Python?
- Wer hat Erfahrung mit Java (Reverse Engineering)?

# Was ist Passwortwiederherstellung?

Passwortwiederherstellung ist der Prozess, der dazu dient ein nicht (mehr) vorhandenes Passwort wiederzuerlangen.

---

## Haftungsausschluss

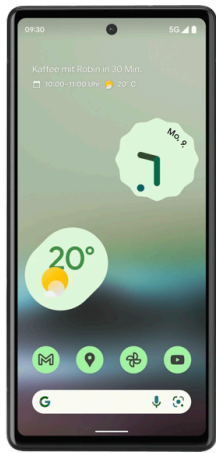
*Wer unbefugt sich oder einem anderen Zugang zu Daten, die nicht für ihn bestimmt und die gegen unberechtigten Zugang besonders gesichert sind, unter Überwindung der Zugangssicherung verschafft, wird mit Freiheitsstrafe bis zu drei Jahren oder mit Geldstrafe bestraft.*

—§ 202 a Abs. 1 StGB

## Gericht sieht Nutzung von Klartext-Passwörtern als Hacken an

*[...] Vor dem Amtsgericht wurde ein Prozess verhandelt, der die Gefahren verdeutlicht, denen sich Menschen mitunter aussetzen, die versuchen, Sicherheitslücken in der Software deutscher Firmen zu finden. Das Amtsgericht hat einen Programmierer verurteilt, der im Auftrag eines Kunden eine Software analysiert und darin eine Sicherheitslücke gefunden hatte, welche die Daten von Einkäufern in Online-Shops im Internet offengelegt hatte. Der Programmierer kontaktierte [...] die betroffene Firma, die*

# Ist Passwortwiederherstellung noch relevant?



---

Klassische Passwörter werden (noch immer) in zahlreichen Bereichen verwendet. Beispiele sind Smartphones, Cryptosticks, Logins für Computer und Serversysteme, verschlüsselte Dateien und Datenträger.

## Hintergrund

Obwohl an vielen Stellen versucht wird Passwörter aus vielen Gründen zurück zu drängen, so ist die Verwendung noch allgegenwärtig und in machen Bereichen ist auch nicht unmittelbar eine Ablösung zu erkennen.

Biometrie ist zum Beispiel in machen Bereichen kein Ersatz für Passwörter und wird - wenn überhaupt - nur ergänzend genommen. So ist es zum Beispiel im deutschen Recht erlaubt/möglich einem Beschuldigten sein Smartphone bei Bedarf vor das Gesicht zu halten, um es zu entsperren. Je nach Qualität des Fingerabdrucksensors können ggf. auch genommene Fingerabdrücke verwendet werden. Möchte der Beschuldigte jedoch das Passwort nicht freiwillig nennen, dann besteht keine direkte weitere Handhabe.

# Ist Passwortwiederherstellung noch relevant?

*Microsoft said hackers working for the Russian government breached its corporate networks recently and stole email from executives and some employees to find out what the company knew about them. The tech company said the breach was not due to any flaw in its software, but rather began with a “password spraying.” The technique worked on what Microsoft said was an old test account, and the hackers then used the account’s privileges to get access to multiple streams of email.*

*—The Washington Post; Joseph Menn (January 19, 2024)*

# Ist Passwortwiederherstellung noch relevant?

## *Researchers Uncover How Outlook Vulnerability Could Leak Your NTLM Passwords*

*A now-patched security flaw in Microsoft Outlook could be exploited by threat actors to access NT LAN Manager (NTLM) v2 hashed passwords when opening a specially crafted file.*

*[...] Varonis security researcher Dolev Taler, who has been credited with discovering and reporting the bug, said NTLM hashes could be leaked by leveraging Windows Performance Analyzer (WPA) and Windows File Explorer. These two attack methods, however, remain unpatched.*

*"What makes this interesting is that WPA attempts to authenticate using NTLM v2 over the open web," Taler said.*

*—The Hacker News (Jan 29, 2024)*

# Ist Passwortwiederherstellung nicht „trivial“?



An AI just cracked your password.

Home Security Heroes

# Wiederherstellung von Passwörtern mit unterschiedlicher Komplexität

Beurteilen Sie die Qualität der folgenden Passwörter in Hinblick darauf wie aufwändig es ist das Passwort wiederherzustellen:

1. Donaudampfschiffahrt
2. Passwort
3. ME01703138541
4. 2wsx3edc4rfv
5. Haus Maus
6. iluvu
7. Emily18
8. MuenchenHamburg2023!!!!
9. hjA223dn4fw"üäKßß k`≤-~ajsdk
10. Baum Lampe Haus Steak Eis Berg
11. password123

# Quellen für Passwortkandidaten

- Wörterbücher
- Verzeichnisse (z.B. Postleitzahlen, Städte, Straßennamen)
- Leaks (Sammlungen von realen Passwörtern, die meist von Hackern veröffentlicht wurden.)
  - Rockyou
  - LinkedIn
  - Sony
  - etc.



# Raum der Passwortkandidaten

- Eine vierstellige PIN: 10.000 mögliche Kombinationen.
- „Normales“ Passworte mit 8 Zeichen und 70 Zeichen im Zeichensatz (a-z, A-Z, 0-9 und ausgewählte Sonderzeichen):  $70^8 = 576.480.100.000.000$  Kombinationen.
- Eine einfache Passphrase mit 4 Wörtern aus einem Wörterbuch mit 100.000 Wörtern:  $100.000^4 = 10^{20}$  Kombinationen.
- Ein komplexes Passwort mit 16 Zeichen und 84 Zeichen im Zeichensatz (a-z, A-Z, 0-9 und die meisten Sonderzeichen):  $84^{16} = 6,14 \times 10^{30}$  Kombinationen.

Eine vierstellige PIN kann niemals als sicher angesehen werden. Selbst wenn ein Brute force nur auf 4 oder 5 Versuche pro Stunde kommt, so ist es dennoch in wenigen Monaten möglich die PIN zu ermitteln.

## **Warnung**

Es ist nie eine Option Passwörter im Klartext zu speichern.

# 1. KRYPTOGRAFISCHE HASHFUNKTIONEN UND PASSWÖRTER

Prof. Dr. Michael Eichberg

# Hashfunktionen (Wiederholung)

- Eine Hashfunktion  $H$  akzeptiert eine beliebig lange Nachricht  $M$  als Eingabe und gibt einen Wert fixer Größe zurück:  $h = H(M)$ .
- Eine Änderung eines beliebigen Bits in  $M$  sollte mit hoher Wahrscheinlichkeit zu einer Änderung des Hashwerts  $h$  führen.
- Kryptographische Hashfunktionen werden für die Speicherung von Passwörtern verwendet.

## Kollisionen bei Hashes

Wenn ein Passwort „nur“ als Hash gespeichert wird, dann gibt es zwangsläufig Kollisionen und es könnte dann theoretisch passieren, dass ein Angreifer (zufällig) ein völlig anderes Passwort findet, dass bei der Überprüfung des Passworts akzeptiert wird. Die Konstruktion kryptografischer Hashfunktionen stellt jedoch sicher, dass dies in der Praxis nicht auftritt.

# Kryptografische Hashfunktionen für Passworte

- Bekannte kryptografische Hash-Funktionen: MD4, MD5, SHA-256, SHA-512, RIPE-MD, ...
- Bekannte Funktion zur Schlüsselableitung: PBKDF2, ...
- Beim Hashing von Passwörtern werden die Basisalgorithmen in der Regel mehrfach (ggf. viele hunderttausend Male) angewendet, um die Laufzeit zu verlängern und es für Angreifer schwieriger zu machen.
- Mehrere Hash-Algorithmen/Schlüsselableitungsfunktionen wurden ausdrücklich für das Hashing von Passwörtern entwickelt, um gängigen Angriffen zu widerstehen. z.B. bcrypt, scrypt, Argon2.
- Einige dieser Algorithmen sind so rechenintensiv, dass sie nicht für Webanwendungen bzw. Situationen geeignet sind, in denen viele Benutzer gleichzeitig autorisiert werden müssen. Diese Algorithmen werden in der Regel zum Schutz von Dateien, Containern oder lokaler Festplatten verwendet.

# Vom Salzen ( *Salt*) ...

## Beobachtung/Problem

Werden Passwörter direkt mit Hilfe einer kryptografischen Hashfunktion gehasht, dann haben zwei Nutzer, die das gleiche Passwort verwenden, den gleichen Hash.

| User  | Hash   |
|-------|--|
| Alice | <code>sha256_crypt.hash('DHBWMannheim',salt='',rounds=1000) = \$5\$rounds=1000\$\$lb/CwYgN/xR9dqYuYxYVtWkxMEh.VK.Q0C9IKmy9DP/</code> |
| Bob   | <code>sha256_crypt.hash('DHBWMannheim',salt='',rounds=1000) = \$5\$rounds=1000\$\$lb/CwYgN/xR9dqYuYxYVtWkxMEh.VK.Q0C9IKmy9DP/</code> |


## Lösung

Passwörter sollten immer mit einem einzigartigen und zufälligen „Salt“ gespeichert werden, um Angriffe mittels Regenbogentabellen zu verhindern.

| User  | Hash   |
|-------|--|
| Alice | <code>sha256_crypt.hash('DHBWMannheim',salt='0123456',rounds=1000) \$5\$rounds=1000\$0123456\$66x8MB.qev25coq90VrD1lr1ZGJJelAz0VlCDZykrY0</code> |
| Bob   | <code>sha256_crypt.hash('DHBWMannheim',salt='1234567',rounds=1000) \$5\$rounds=1000\$1234567\$LxD/hg29N9KYpNdFMW69Kk65BLkVLLzLSEJvqhCmFU9</code> |

14

## Regenbogentabellen

Eine Regenbogentabelle ( *rainbow table*) bezeichnet eine vorberechnete Tabelle die konzeptionell zum einem Hash ein jeweilig dazugehörendes Passwort speichert und einen effizienten Lookup ermöglicht. Dies kann ggf. die Angriffsgeschwindigkeit sehr signifikant beschleunigen.

Aufgrund der allgemeinen Verwendung von Salts sind Angriffe mit Hilfe von Regenbogentabellen heute nur noch von historischer Bedeutung.

# Vom Salzen ( *Salt*)...

- Ein *Salt* sollte ausreichend lang sein (zum Beispiel 16 Zeichen oder 16 Byte).
- Ein *Salt* darf nicht wiederverwendet werden.
- Ein *Salt* kann zum Beispiel (am Anfang oder) am Ende an das Passwort angehängt werden bevor selbiges gehasht wird.
- Ein *Salt* unterliegt (eigentlich) keinen Geheimhaltungsanforderungen.

## Problem

Sollte es einem Angreifer gelingen in eine Datenbank einzubrechen und die Tabellen mit den Nutzerdaten abzufragen (zum Beispiel aufgrund einer erfolgreichen SQL Injection), dann ist es ihm danach direkt möglich zu versuchen Passworte wiederherzustellen.

## Speicherung von Salts

In Webanwendungen bzw. allgemein datenbankgestützten Anwendungen wird der *Salt* häufig in der selben Tabelle gespeichert in der auch der Hash des Passworts gespeichert wird. Im Falle von verschlüsselten Dateien, wird der Salt (unverschlüsselt) mit in der Datei gespeichert.

## ... und Pfeffern ( *Pepper*) von Passwörtern

(In Normen/Teilen der Literatur wird statt *Pepper* auch *Secret Keys* verwendet.)

- Wie ein *Salt* geht auch der *Secret Key* in den Hashvorgang des Passworts ein.
- Der *Secret Key* wird jedoch **nicht** mit den Hashwerten der Passworte gespeichert.
- Ein *Secret key* kann zum Beispiel in einem Hardwaresicherheitsmodul (z.B. Secure Element oder TPM Chip) gespeichert werden. Gel. wird der *Secret Key* bzw. ein Teil davon auch im Code gespeichert.
- Wie ein Salt sollte auch auch *Secret Key* mind. 16 Byte lang sein, um ggf. ein Brute-Force Angriff auf den *Secret Key* zu verhindern sollte dem Angreifer zu einem Hash und Salt auch noch das Klartext Passwort bekannt sein.
- Der *Secret Key* sollte zufällig sein.
- Der *Secret Key* sollte pro Instanziierung einer Anwendung einmalig sein.



# Sichere Hashfunktionen für Passworte

- Argon2 (z.B. verwendet von LUKS2)
- bcrypt (basierend auf Blowfish)
- scrypt (z.B. ergänzend verwendet für das Hashing von Passwörtern auf Smartphones)
- yescrypt (z.B. modernen Linux Distributionen)

# PBKDF2 (Password-Based Key Derivation Function 2)

- Dient der Ableitung eines Schlüssels aus einem Passwort.
- Das Ergebnis der Anwendung der PBKDF2 wird zusammen mit dem *Salt* und dem Iterationszähler für die anschließende Passwortverifizierung gespeichert.
- die *PBKDF2* Schlüsselableitungsfunktion hat 5 Parameter  $DK = PBKDF2(\text{PRF}, \text{Password}, \text{Salt}, c, \text{dkLen})$ :

|                  |   |
|------------------|---|
| <b>PRF:</b>      | Eine Pseudozufallsfunktion; typischer Weise ein HMAC. |
| <b>Password:</b> | Das Masterpasswort.                                   |
| <b>Salt:</b>     | der zu verwendende Salt.                              |
| <b>c:</b>        | Zähler für die Anzahl an Runden.                      |
| <b>dkLen:</b>    | Die Bitlänge des abgeleiteten Schlüssels.             |

Die PBKDF2 ist nicht für das eigentliche Hashen zuständig sondern „nur“ für das Iterieren der Hashfunktion und das eigentliche Key-stretching.

Laut OWASP sollten zum Beispiel für PBKDF2-HMAC-SHA512 600.000 Iterationen verwendet werden.

# PBKDF2-HMAC (Hash-based Message Authentication Code)

Im Fall von PBKDF2 ist der Schlüssel  $K$  also das Passwort und die Nachricht  $M$  das Salz.

## Beispielcode

```
from passlib.crypto.digest import pbkdf2_hmac
pbkdf2_hmac("sha256",
    secret=b"MyPassword",
    salt=b"JustASalt",
    rounds=1,    # a real value should be >> 500.000
    keylen=32 )
```

Bei einer Runde und passenden Blockgrößen ist das Ergebnis der PBKDF2 somit gleich mit der Berechnung des HMACs wenn der Salt um die Nummer des Blocks `\x00\x00\x00\x01` ergänzt wurde.

In der konkreten Anwendung ist es ggf. möglich das *Secret* auch zu Salzen und den *Salt* aus einer anderen Quellen abzuleiten.

## Schwachstellenbewertung

Ihnen liegt eine externer Festplatte/SSD mit USB Anschluss vor, die die folgenden Eigenschaften hat:

- Die Daten auf der SSD/FP sind hardwareverschlüsselte Festplatte
- Die Verschlüsselung erfolgt mit XTS-AES 256
- Es gibt eine spezielle Software, die der Kunde installieren muss, um das Passwort zu setzen. Erst danach wird die Festplatte „freigeschaltet“ und kann in das Betriebssystem eingebunden werden. Davor erscheint die SSD/FP wie ein CD Laufwerk auf dem die Software liegt.
- Die SSD/FP ist FIPS zertifiziert und gegen Hardwaremanipulation geschützt; zum Beispiel eingegossen mit Epox.
- Das Passwort wird von der Software gehasht und dann als Hash an den Controller der externen FP/SSD übertragen.
- Im Controller wird der übermittelte Hash direkt zur Autorisierung des Nutzers verwendet. Dazu wird der Hash mit dem im EPROM hinterlegten verglichen.

Wie bewerten Sie die Sicherheit des Produkts?

## 2. PASSWORTWIEDERHERSTELLUNG 101

Prof. Dr. Michael Eichberg

# Passwortwiederherstellung

1. Wissen wo/in welcher Form der Passworthash zu finden ist.
2. Extraktion des Hashes
3. Wiederherstellung des Passwortes durch das systematische Durchprobieren aller Kandidaten.

# Beispiel - Wiederherstellung eines Linux Login Passwortes

```
~% sudo cat /etc/shadow
[...]
john:$6$zElzjLsMqi36JXWG$FX2Br1/[...]. ↵
RxAHnNCBsqiouWUz751crHodXxs0iqZfBt9j40l3G0:19425:0:99999:7:::
[...]
```

```
% echo -n '$6$zElzjLsMqi36JXWG$FX2Br1/[...]. ↵
RxAHnNCBsqiouWUz751crHodXxs0iqZfBt9j40l3G0' > hash.txt
```

```
% hashcat -m 1800 hash.txt -a 3 '?d?d?d?d?d?d'
```

23

## Finden eines Hashes

Im Falle von Linux Login Passworten ist genau spezifiziert wo die Passworte (**/etc/shadow**) und in welcher Form die Passworte gespeichert werden. Nach dem Namen des Nutzers (im Beispiel **john**) ist der verwendete Hashingalgorithmus vermerkt. Dieser unterscheidet sich zwischen den Distributionen. Aktuell setzen die meisten Distributionen auf **yescrypt**. Danach folgen die Parameter. Insbesondere der Salt.

| ID               | Mode                      |
|------------------|---------------------------|
| \$5\$            | Sha256crypt (veraltet)    |
| \$6\$            | SHA512crypt (in Ablösung) |
| \$y\$ (or \$7\$) | yescrypt                  |

# Systematisches Testen aller Kandidaten

konzeptionell führt die Software Hashcat die folgenden Schritte durch:

```
<extracted_hash> =? SHA512crypt("zElzjLsMqi36JXWG","000000") x
<extracted_hash> =? SHA512crypt("zElzjLsMqi36JXWG","000001") x
<extracted_hash> =? SHA512crypt("zElzjLsMqi36JXWG","000002") x
<extracted_hash> =? SHA512crypt("zElzjLsMqi36JXWG","000003") x
<extracted_hash> =? SHA512crypt("zElzjLsMqi36JXWG","000004") x
<extracted_hash> =? SHA512crypt("zElzjLsMqi36JXWG","000005") x
<extracted_hash> =? SHA512crypt("zElzjLsMqi36JXWG","000006") x
<extracted_hash> =? SHA512crypt("zElzjLsMqi36JXWG","000007") x
<extracted_hash> =? SHA512crypt("zElzjLsMqi36JXWG","000008") x
...
<extracted_hash> == SHA512crypt("zElzjLsMqi36JXWG","123456") ✓
```

24

Der folgende Code könnte als Grundlage genutzt werden, um das Passwort wiederherzustellen. (Linux nutzt standardmäßig 5000 Runden.)

```
from passlib.hash import sha512_crypt

sha512_crypt.hash("123456", salt="zElzjLsMqi36JXWG", rounds=5000)
```




# 3. PASSWORTE VERSTEHEN

Prof. Dr. Michael Eichberg

# Aufbau von Passwörtern

Von Menschen vergebene Passwörter basieren häufig auf Kombinationen von Wörtern aus den folgenden Kategorien:

- Pins: 1111, 1234, 123456, ...
- Tastaturwanderungen ( *keyboard walks*): **asdfg**, **q2w3e4r5t**, ...
- Patterns: aaaaaa, ababab, abcabcabc, ...
- Reguläre Wörter aus Wörterbüchern: Duden, Webster, ...
- Kontextinformationen:
  - Szenespezifisch: **acab**, ...
  - Privates Umfeld: Namen von Kindern, Eltern, Hunden, Geburtsort, Adresse, ...

# Häufige Passworte

Eine gute Quelle für das Studium von Passwörtern sind sogenannte *Leaks* oder auch Listen mit gängigen Passwörtern. Zum Beispiel **Becker's Health IT 2023**:

|            |           |            |
|------------|-----------|------------|
| 123456     | abc123    | princess   |
| password   | 1234      | letmein    |
| 123456789  | password1 | 654321     |
| 12345      | iloveyou  | monkey     |
| 12345678   | 1q2w3e4r  | 27653      |
| qwerty     | 000000    | 1qaz2wsx   |
| 1234567    | qwerty123 | 123321     |
| 111111     | zaq12wsx  | qwertyuiop |
| 1234567890 | dragon    | superman   |
| 123123     | sunshine  | asdfghjkl  |

## Hinweise

- Die Listen ändern sich in der Regel von Jahr zu Jahr nicht wesentlich.
- Die Methodik ist oft fragwürdig.

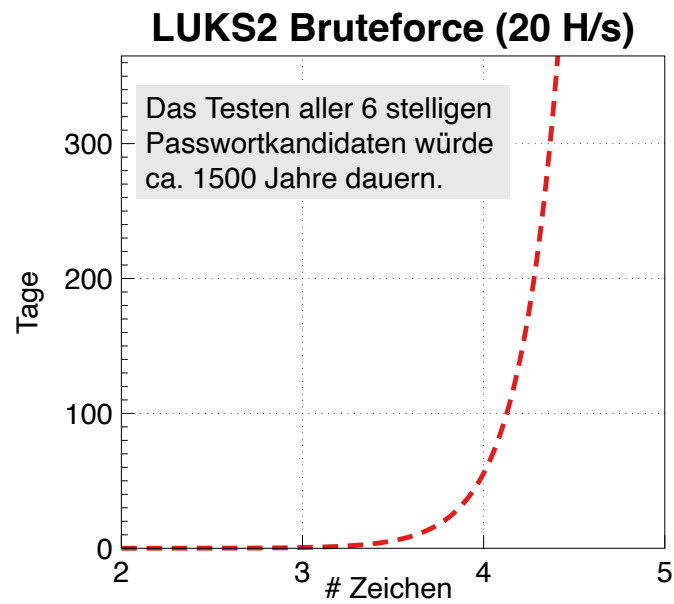
# Herausforderung: Hashraten in MH/s auf aktueller Hardware

| Hashcat Mode<br>(Hashcat 6.2.6) | Hash                                 | RTX 1080Ti<br>250 W | RTX 2080Ti<br>260 W | RTX 3090<br>350 W | RTX 4090<br>450 W      |
|---------------------------------|--------------------------------------|---------------------|---------------------|-------------------|------------------------|
| 25700                           | Murmur                               |                     |                     |                   | 643700.0<br>(643 GH/s) |
| 23                              | Skype                                | 21330.1             | 27843.1             | 37300.7           | 84654.8                |
| 1400                            | SHA2-256                             | 4459.7              | 7154.8              | 9713.2            | 21975.5                |
| 10500                           | PDF1.4-1.6                           | 24.9                | 29.8                | 76.8              | 122.0                  |
| 1800                            | SHA 512 Unix (5000<br>Iterations)    | 0.2                 | 0.3                 | 0.5               | 1.2                    |
| 13723                           | Veracrypt SHA2- 512 +<br>XTX 1536Bit | 0.0004              | 0.0006              | 0.0009            | 0.002 (2000<br>H/s)    |

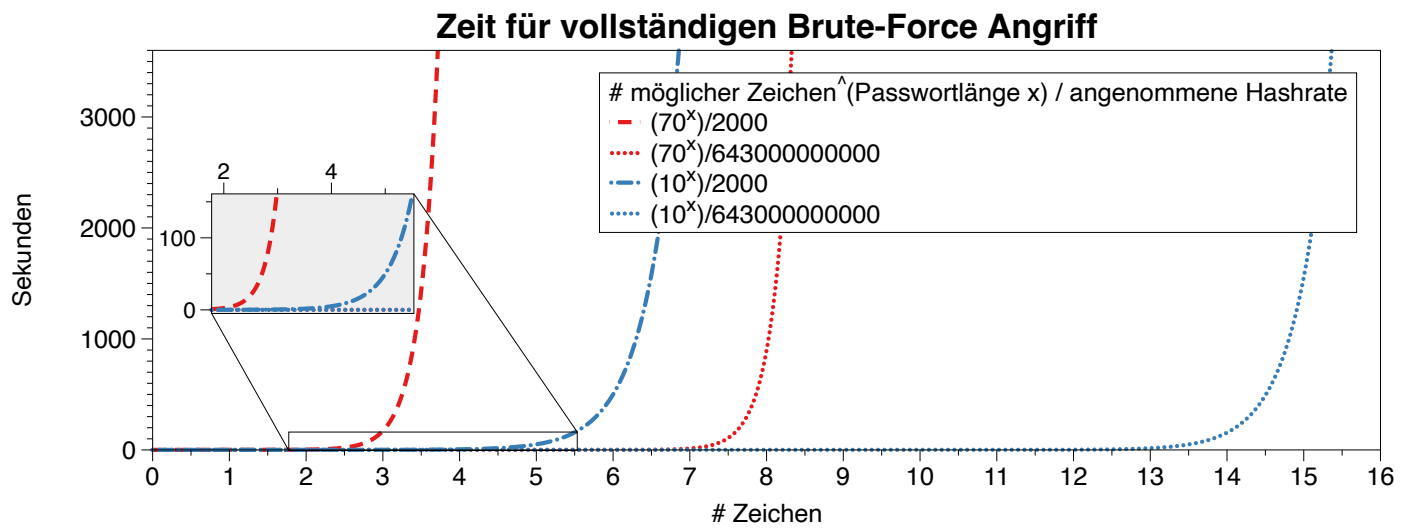
## Quellen:

**4090:** <https://gist.github.com/Chick3nman/e4fcee00cb6d82874dace72106d73fef>  
**3090:** <https://gist.github.com/Chick3nman/e4fcee00cb6d82874dace72106d73fef>  
**1080Ti:** <https://www.onlinehashcrack.com/tools-benchmark-hashcat-nvidia-gtx-1080-ti.php>  
**2080Ti:** <https://gist.github.com/binary1985/c8153c8ec44595fdabbf03157562763e>

# Herausforderung: Unmöglichkeit eines Brute-Force Angriffs auf Luks2



# Herausforderung: Unmöglichkeit eines Brute-Force Angriffs auf lange Passworte



# Herausforderung: stets neue Algorithmen

## *Angriff auf LUKS2 mit Argon2*

*[...] The choice of Argon2 as a KDF makes GPU acceleration impossible. As a result, you'll be restricted to CPU-only attacks, which may be very slow or extremely slow depending on your CPU. To give an idea, you can try 2 (that's right, two) passwords per second on a single Intel(R) Core(TM) i7-9700K CPU @ 3.60GHz. Modern CPUs will deliver a slightly better performance, but don't expect a miracle: LUKS2 default KDF is deliberately made to resist attacks. [...]*

*—Elcomsoft **Luks2 with Argon2***

## Kosten und Aufwand für Passwortwiederherstellung

Sie wollen einen SHA 256 angreifen und sie haben 100 Nvidia 4090 GPUs. Jede GPU hat eine Hash-Rate von ~22GH/s (mit Hashcat 6.2.6) und benötigt ~500 Watt. Der verwendete Zeichensatz besteht aus 84 verschiedenen Zeichen (z.B. a-z, A-Z, 0-9, <einige Sonderzeichen>).

1. Wie lange dauert es, ein 10-stelliges Passwort zu ermitteln (Worst Case)?
2. Wie viel Geld wird es Sie kosten, ein 10-stelliges Passwort zu knacken (Worst Case) (1kW ~ 0,25ct)?
3. Werden Sie im Laufe Ihres Lebens in der Lage sein, ein Passwort mit 12 Zeichen zu ermitteln?



## Verstehen des Suchraums

Sie haben ganz viele Grafikkarten und einen sehr schnellen Hash. Sie kommen auf eine Hashrate von 1THash/Sekunde ( $1 \times 10^{12}$ ). Sie haben einen Monat Zeit für das Knacken des Passworts. Gehen Sie davon aus, dass Ihr Zeichensatz 100 Zeichen umfasst.

Berechnen Sie den Anteil des Suchraums, den Sie abgesucht haben, wenn das Passwort 32 Zeichen lang sein sollte. Drücken Sie den Anteil des abgesuchten Raums in Relation zu der Anzahl der Sandkörner der Sahara aus. Gehen Sie davon aus, dass die Sahara ca. 70 Trilliarden ( $70 \times 10^{21}$ ) Sandkörner hat.

# Herausforderung: Passwortrichtlinien

Moderne Passwortrichtlinien (🚩 *Password Policies*) machen es unmöglich, ältere Leaks *direkt* zu nutzen.

*Beispiele:*

- Mindestanzahl von Zeichen (maximale Anzahl von Zeichen)
- Anforderungen an die Anzahl der Ziffern, Sonderzeichen, Groß- und Kleinbuchstaben
- Anforderungen an die Vielfalt der verwendeten Zeichen
- einige Passwörter (z.B. aus bekannten Leaks und Wörterbüchern) sind verboten
- ...

34

Passwortrichtlinien extrem: **Password Game**

Die wichtigsten **NIST-Richtlinien** für Passwörter:

- Mindestlänge von 8 Zeichen.
- Keine Komplexitätsanforderung. Benutzer sollten auch die Möglichkeit haben, Leerzeichen einzufügen, um die Verwendung von Phrasen zu ermöglichen. Für die Benutzerfreundlichkeit [...] kann es von Vorteil sein, wiederholte Leerzeichen in getippten Passwörtern vor der Überprüfung zu entfernen.

# Die Struktur von Passwörtern verstehen

Analyse auf Grundlage des „berühmten“ Rockyou-Lecks.

Hier haben wir alle Kleinbuchstaben auf l, Großbuchstaben auf u, Ziffern auf d und Sonderzeichen auf s abgebildet.

|            |         |               |         |               |         |            |         |
|------------|---------|---------------|---------|---------------|---------|------------|---------|
| llllllll   | 4,8037% | llllllldd     | 1,4869% | ddddddddddddd | 0,2683% | ddddddll   | 0,1631% |
| llllll     | 4,1978% | llllld        | 1,3474% | llldddd       | 0,2625% | llllls     | 0,1615% |
| llllll     | 4,0849% | llllld        | 1,3246% | llllllllldd   | 0,2511% | dddllll    | 0,1613% |
| llllllll   | 3,6086% | llllllllll    | 1,3223% | llllllllllll  | 0,2340% | dllllll    | 0,1583% |
| ddddddd    | 3,4003% | llldddd       | 1,2439% | llldddd       | 0,2322% | dllllll    | 0,1575% |
| dddddddddd | 3,3359% | llllldddd     | 1,2109% | lldddd        | 0,2270% | llldddd    | 0,1560% |
| ddddddd    | 2,9878% | llllldddd     | 1,1204% | uuuuuudd      | 0,2189% | dddddddl   | 0,1557% |
| llllldd    | 2,9326% | llllld        | 1,1168% | dddll         | 0,2169% | uuuudd     | 0,1551% |
| llllllll   | 2,9110% | llllldd       | 1,0633% | ldddd         | 0,2064% | llllldddd  | 0,1395% |
| dddddd     | 2,7243% | llllldddd     | 0,9225% | ddddddddddddd | 0,2017% | ddl        | 0,1391% |
| dddddddddd | 2,1453% | llllllld      | 0,9059% | ullllld       | 0,1930% | ullll      | 0,1379% |
| llllld     | 2,0395% | lllll         | 0,8793% | dddllll       | 0,1905% | uuuuuuuuuu | 0,1378% |
| llllldd    | 1,9092% | llllllllll    | 0,8334% | uuuuuuuuu     | 0,1886% | llllls     | 0,1374% |
| llllllll   | 1,8697% | llllld        | 0,8005% | uuuuuudd      | 0,1815% | llllllld   | 0,1345% |
| llldddd    | 1,6420% | llldddd       | 0,7759% | llllllldd     | 0,1808% | llllllldd  | 0,1344% |
| llldd      | 1,5009% | ddddddddddddd | 0,7524% | llllllldddd   | 0,1725% | ...        | ...     |

# Die Zusammensetzung von Passwörtern verstehen

Analyse des *rockyou* Leaks.

|                          |                   |             |
|--------------------------|-------------------|-------------|
| <b>Σ Passworte</b>       | <b>14.334.851</b> | <b>100%</b> |
| Pins                     | 2.346.591         | 16,37%      |
| Passworte mit Buchstaben | 11.905.977        | 83,34%      |

Analyse der Passworte mit Buchstaben:

| Kategorie                     | Absolut   | Prozentual    | Beispiele   |
|-------------------------------|-----------|---------------|---|
| Emails                        | 26.749    | 0,22%         | me@me.com   |
| Zahlen gerahmt von Buchstaben | 35696     | 0,30%         | a123456a  |
| Leetspeak                     | 64.672    | 0,54%         | G3tm0n3y  |
| Patterns                      | 124.347   | 1,04%         | lalala  |
| Reguläre oder Populäre Wörter | 4.911.647 | <b>41,25%</b> | princess      iloveu                                |
| Sequenzen                     | 5.290     | 0,04%         | abcdefghij  |
| keyboard walks (de/en)        | 14.662    | 0,12%         | q2w3e4r   |
| Einfache Wortkombinationen    | 535.037   | 4,49%         | pinkpink      sexy4u      te amo                    |
| Komplexe Wortkombinationen    | 5.983.259 | <b>50,25%</b> | Inparadise      kelseylovesbarry                    |
| <Rest>                        | 204.618   | 1,72%         | j4**9c+p      i(L)you      p@55w0rd      sk8er4life |

# Der Effekt von Passwortrichtlinien auf Passwörter

Reale Passwortrichtlinie:

Nutze 1 Großbuchstabe, 1 Kleinbuchstabe, 2 Symbole, 2 Ziffern, 4 Buchstaben, 4 Nicht-Buchstaben

Exemplarisch beobachteter Effekt wenn die Passwörter vorher einfacher waren und der Benutzer gezwungen wurde diese zu erweitern:

Password11##

Password12!!

d.h. die Passworte werden mit möglichst geringem Aufwand erweitert.

# Aufbau von Passwörtern - Zusammenfassung

- Passwörter, die häufig eingegeben werden müssen, basieren in den allermeisten Fällen auf „echten“ Wörtern.
- Echte Wörter werden oft nicht unverändert verwendet, sondern nach einfachen Regeln umgewandelt, z.B. durch Anhängen einer Zahl oder eines Sonderzeichens, Veränderung der Groß-/Kleinschreibung, etc.

## Frage

Wie können wir gute Passwortkandidaten identifizieren/generieren, wenn ein *Leak* nicht ausreicht oder nur eine kleine Anzahl von Passwörtern getestet werden kann?

Zum Beispiel dauert das Testen aller Passwörter von Rockyou...:

~13.000.000 Passworte / 5 Hashes/Sekunde  $\approx$  1 Monat

~13.000.000 Passworte / 5 Hashes/Stunde  $\approx$  ~297 Jahre

# Herausforderungen beim Testen/Generieren von Passwörtern


Aufgrund der „Unmöglichkeit“ eines Brute-Force-Angriffs stellen sich folgende Herausforderungen:

- Verfügbare *Kontextinformationen* sollten in die Auswahl/Generierung einfließen.
- Es sollten nur *technisch sinnvolle* Passwörter getestet/generiert werden.
- Es sollten *keine Duplikate* getestet werden.
- Auswahl/Generierung von *Passwörtern in absteigender Wahrscheinlichkeit*.
- Die Auswahl/Generierung sollte effizient sein.

Technisch sinnvolle Passwörter sind solche, die die zu Grunde liegenden Passwortrichtlinien und auch weiteren technischen Anforderungen erfüllen. Zum Beispiel den von der Software verwendeten Zeichensatz (UTF-8, ASCII, ...) oder im Falle eines Smartphones/Kryptosticks die eingebbaren Zeichen.

Sollte der Algorithmus zum Generieren der Passwörter langsamer sein als die Zeit, die benötigt wird, um ein Passwort zu falsifizieren, dann beschränkt nicht mehr länger nur die Hashrate den Suchraum.

# Ansätze und Werkzeuge zum Generieren von Passwortlisten

- Grundlegende Werkzeuge zum „vermischen von Wörtern“ ( *Word-mangling*)
  - Prince
  - Markov-Modelle (OMEN)
  - Hashcat
  - ...

Um vorhandene Kontextinformationen zu erweitern, können ggf. (frei) verfügbare Wordembeddings verwendet werden.

- [RelatedWords.org](https://relatedwords.org/) setzt (unter anderem) auf ConceptNet und WordEmbeddings.
- [Reversedictionary.org](https://reversedictionary.org/) setzt auf WordNet und liefert ergänzende Ergebnisse.



# Markov-Ketten

OMEN lernt - zum Beispiel basierend auf Leaks - die Wahrscheinlichkeiten für das Aufeinanderfolgen von Bigrammen und Trigrammen und nutzt diese, um neue Passwortkandidaten zu generieren.

## Hintergrund

Eine Markov-Kette beschreibt eine Sequenz möglicher Ereignisse in welcher die Wahrscheinlichkeit des Nächsten nur vom Zustand des vorherigen abhängt.

## Grundlegende Idee

Gegeben: `lachen`, `Sachen`, `Last`, `Muster`

Bigramme: `2*la`, `2*ch`, `2*en`, `sa`, `2*st`, `mu`, `er`

Auf ein `st` folgt entweder ein `er` oder `<Wortende>`; demzufolge ist `laster` ein Kandidat, aber auch `must`.

# Password Cracking Using Probabilistic Context-Free Grammars **[PCFG]**

- Lernt die Muster, Worte, Ziffern und verwendeten Sonderzeichen basierend auf der Auswertung von realen Leaks. Die gelernte Grammatik wird als Schablone verwendet und aus „Wörterbüchern“ befüllt. (Zum Beispiel:  $S \rightarrow D1L3S2 \rightarrow 1L3!! \rightarrow 1luv!!$  )
- Generiert Passwortkandidaten mit absteigender Wahrscheinlichkeit.
- Prozeß:
  1. Vorverarbeitung, um die Basisstrukturen und deren Wahrscheinlichkeiten zu identifizieren (z.B. zwei Ziffern gefolgt von einem Sonderzeichen und 8 Buchstaben.)
  2. Passwortkandidatengenerierung unter Beachtung der Wahrscheinlichkeiten der Basisstrukturen und der Wahrscheinlichkeiten der Worte, Ziffern und Sonderzeichen.  
(In der Originalversion wurden die Wahrscheinlichkeiten von Worten nicht beachtet; die auf GitHub verfügbare Version enthält jedoch zahlreiche Verbesserungen.)`

# PCFG - Analyse - Beispiel

Im ersten Schritt werden die Produktionswahrscheinlichkeiten von Basisstrukturen, Ziffernfolgen, Sonderzeichenfolgen und Alpha-Zeichenfolgen ermittelt. (Z. Bsp.: !cat123  $\Rightarrow$  S<sub>1</sub>L<sub>3</sub>D<sub>3</sub>)

| Basis Struktur | Häufigkeit | Wahrscheinlichkeit der Produktion |
|----------------|------------|-----------------------------------|
| L3S1D3         | 12788      | 0.75                              |
| S1L3D3         | 2789       | 0.35                              |

| S1 | Häufigkeit | Wahrscheinlichkeit der Produktion |
|----|------------|-----------------------------------|
| !  | 12788      | 0.50                              |
| .  | 2789       | 0.30                              |
| @  | 1708       | 0.20                              |

| L3  | Häufigkeit | Wahrscheinlichkeit der Produktion |
|-----|------------|-----------------------------------|
| cat | 12298      | 0.85                              |
| dog | 2890       | 0.15                              |

| D3  | Häufigkeit | Wahrscheinlichkeit der Produktion |
|-----|------------|-----------------------------------|
| 123 | 10788      | 0.60                              |
| 321 | 5789       | 0.35                              |
| 654 | 4708       | 0.25                              |

# PCFG - Generierung - Beispiel

Ergebnis der Analyse:

| Nich-Terminal | Produktion | Wahrscheinlichkeit der Produktion |
|---------------|------------|-----------------------------------|
| S             | passwordT  | 0.7                               |
| S             | secureT    | 0.3                               |
| T             | 123        | 0.6                               |
| T             | 111        | 0.4                               |

## Hinweis

Nicht-Terminal = [S,T]

Terminal = [a, b, c, d, e, ..., z, 0, ..., 9]

Ableitung:

1.  $S \Rightarrow \text{passwordT} \Rightarrow \text{password123}$
2.  $S \Rightarrow \text{passwordT} \Rightarrow \text{password111}$
3.  $S \Rightarrow \text{secureT} \Rightarrow \text{secure123}$
4.  $S \Rightarrow \text{secureT} \Rightarrow \text{secure111}$

# PCFG+

## Next Gen PCFG Password Cracking [NGPCFG]:

Unterstützt Tastaturwanderungen (zum Beispiel asdf oder qwerty12345), Passworte bestehend aus mehreren Worten und wiederholten Worten (zum Beispiel qqqpppq).

---

## On Practical Aspects of PCFG Password Cracking [PAofPCFG]:

Im Wesentlichen Performanceoptimierungen, um PCFG schneller zu machen.

---

## Using personal information in targeted grammar-based probabilistic password attacks [PlandPCFG]:

Im Wesentlichen werden zwei PCFGs gewichtet zusammengeführt ( $0 < \alpha < 1$ ).

# SePass: Semantic Password Guessing Using k-nn Similarity Search in Word Embeddings [SePass]

Zusätzliche Wortkandidaten werden mithilfe von *Worteinbettungen* identifiziert.  
Ermöglicht es, automatisch verwandte Wörter zu finden.

## Example

Gegeben:

Ferrari01  
!Audi!  
Mercedes88  
Bugatti 666

"Offensichtliche" Kandidaten für Basiswörter:

Porsche  
Mclaren  
Lamborghini  
Aston Martin

# SePass: Semantic Password Guessing Using k-nn Similarity Search in Word Embeddings

Vermeidet menschliche Voreingenommenheit.

## Example

Gegeben:

Luke2017

John1976

01Mark!

"Offensichtliche" Kandidaten für Basiswörter:

Matthew

Bible

Gospel

# SePass: Semantic Password Guessing Using k-nn Similarity Search in Word Embeddings

Vermeidet menschliche Voreingenommenheit.

## Example

Gegeben:

Luke2017

John1976

01Mark!

"Offensichtliche" Kandidaten für Basiswörter:

Leia

Darth Vader

Palpatine



# Bewertung von Passworten

## **Donaudampfschiffahrt:**

Ist weder in Rockyou noch im Duden und auch nicht in den Corpora von Twitter und Facebook von 2022 zu finden.

**Passwort:** Nr. 93968 in Rockyou.

**password123:** Nr. 1348 in Rockyou.

**2wsx3edc4rfv:** So nicht in Rockyou, aber 1qaz2wsx3edc4rfv ist Nr. 143611 in Rockyou.

**Haus Maus:** In Rockyou ist lediglich hausmaus zu finden.

**iluvu:** Nr. 1472 in Rockyou.

**Emily060218:** Emily ist Nr. 35567 in Rockyou. Die Zahl ist ganz offensichtlich ein Datum: 6. Feb. 2018 und könnte ein Geburtsdatum, Hochzeitsdatum, oder ein für die Person vergleichbar bedeutends Datum sein.

## **MuenchenHamburg2023!!!!:**

Das Passwort ist zwar sehr lang aber es handelt sich vermutlich um zwei - für die entsprechende Person - bedeutende Orte. Die Zahl und die Sonderzeichen sind vermutlich auf eine Passwortrichtlinie zurückzuführen.

## **hJA223dn4fw"üäKBB k`≤~ajsdk:**

28 Stellen basierend auf einem Zeichensatz, der vermutlich ca. 192 Zeichen pro Stelle umfasst.

## **Baum Lampe Haus Steak Eis Berg:**

Es handelt sich um ein Passwort mit 30 Stellen, dass vermutlich mit Hilfe von Diceware generiert wurde und 6 Worte umfasst.

**ME01703138541:** Namenskürzel und Telefonnummer.

49

## **Diceware**

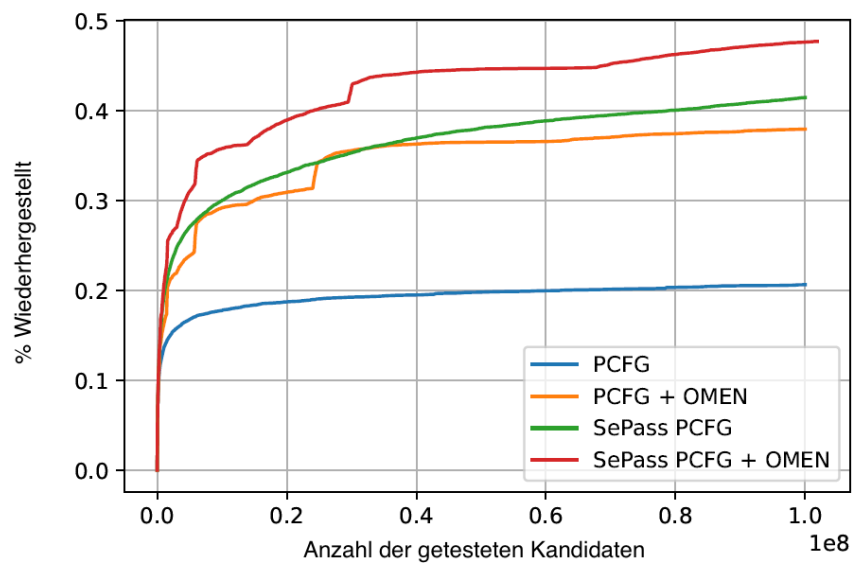
Auch wenn dem Angreifer (a) bekannt ist, dass das Passwort mit Hilfe von Diceware generiert wurde, (b) die zugrundeliegende Wortliste vorliegt und (c) auch die Länge (hier 6 Worte) bekannt sein sollte, dann umfasst der Suchraum:  $(6^5)^6 \approx 2,21 \times 10^{23}$  Passwortkandidaten. Sollte man also mit einer Geschwindigkeit von 1 Billion Hashes pro Sekunde angreifen können, dann braucht man noch immer über 7000 Jahre für das Absuchen des vollständigen Suchraums.

Beim klassischen Dicewareansatz umfasst das Wörterbuch  $6^5$  Worte, da man mit einem normalen Würfel fünfmal würfelt und dann das entsprechende Wort nachschlägt. Würde man zum Beispiel die folgenden Zahlen würfeln: 1,4,2,5,2. Dann würde man das Wort zur Zahl: 14252 nachschlagen.

## **Zeichensatz**

Auf einer deutschen Standardtastatur für Macs können in Kombination mit „Shift“, „Alt“ und „Alt+Shift“ zum Beispiel 192 verschiedene Zeichen eingegeben werden ohne auf Unicode oder Zeichentabellen zurückgreifen zu müssen.

# Wörterbuchgenerierung - Evaluation von Werkzeugen



## **4. WERKZEUGE UND METHODEN ZUR WIEDERHERSTELLUNG VON PASSWÖRTERN**

Prof. Dr. Michael Eichberg

# Grundlegende Werkzeuge

- Linux Shell
- Reguläre Ausdrücke

## Extraktion von Hashes

### **Hinweis**

Im Folgenden diskutieren wir nur exemplarisch die Extraktion einiger Hashes, um das grundlegende Vorgehen zu besprechen. Im Allgemeinen gibt es für weit(er) verbreitete Software häufig bereits Lösungen zur Hashextraktion. Falls nicht, dann muss man Googeln und/oder Reverse Engineering betreiben.

# Quellen für Werkzeuge & Anleitungen

- Hashcat Tools
- John (the Ripper)
- Googeln

# Relevante Linux Kommandozeilenwerkzeuge

- **file** dient der Ermittlung des Typs einer Datei.
- **binwalk** durchsucht Binärdateien in Hinblick auf das Vorkommen bekannter Muster (insbesondere Dateiheader, aber auch Kryptokonstanten etc.)
  - E kann zur Visualisierung der Entropie verwendet werden.
- **dd** kopiert Daten blockweise von einem Startpunkt in einer Datei in eine andere Datei. Wird ggf. zum Extrahieren von Hashes benötigt.
- **xxd** und **hexdump** erstellen beide einen Hexdump einer Datei.

# Verschlüsselte PDF Dateien

Extraktion erfolgt (zum Beispiel) mit den John Tools:

```
$ pdf2john Document.pdf > Document.pdf.john
$ cat Document.pdf.john
Document.pdf:$pdf$4*4*128*-3392*1*16*861da8b9c1672ddc3953dee025
5d622d*32*301d0810078c5698ab17b286e212307000000000000000000000000
000000000000*32*c038ddb8fbdaeb67b6e80e2d936108fc851ff40c5b652c71
97bda4f797939532
```

Danach kann der Hash entweder direkt mit John angegriffen werden, oder nach dem Entfernen des Headers mit Hashcat.

```
$ pdf2john Document.pdf \
| sed -E "s/^[^:]+: //"          # Dateiname entfernen
> Document.pdf.hashcat
```



# Libreoffice Dateien

Extraktion des Basishashes erfolgt auch hier (zum Beispiel) mit den John Tools. Danach muss sowohl der Prefix als auch der Suffix, der für die Entschlüsselung nicht relevant ist, abgeschnitten werden, wenn im Folgenden Hashcat verwendet werden soll.

```
$ libreoffice2john Document.odt  
| sed -E -e 's/[^:]+: //' -e 's/:::~::~[^\:]+$/'  
> Document.odt.hashcat
```

Um zu verstehen, wie der Hash genau auszusehen hat, ist es im Allgemeinen hilfreich sich die erwartete Struktur für einen Hash anzusehen: [Hashcat - Example Hashes](#)

# Verschlüsselte Mac Disk Images (.dmg)

In diesem Fall hat nur John (the Ripper) Unterstützung für den konkreten Hash.

```
$ dmg2john Container.dmg > Container.dmg.john    # Extraktion  
$ john Container.dmg.john \                      # Angriff  
  --wordlist=/usr/share/wordlists/rockyou.txt
```

# Verschlüsselter USB Stick (APFS Volume)

Es liegt ein normaler USB Stick vor auf dem eine Partition vom Typ **Apple APFS** ist.

Disk /dev/sda: 14.45 GiB, 15518924800 bytes, 30310400 sectors

Disk model: Flash Disk

Units: sectors of 1 \* 512 = 512 bytes

Sector size (logical/physical): 512 bytes / 512 bytes

I/O size (minimum/optimal): 512 bytes / 512 bytes

Disklabel type: gpt

Disk identifier: 1D63D8AE-7CBC-47BE-9093-8469B0786EAF

| Device    | Start  | End      | Sectors  | Size  | Type       |
|-----------|--------|----------|----------|-------|------------|
| /dev/sda1 | 40     | 409639   | 409600   | 200M  | EFI System |
| /dev/sda2 | 409640 | 30310359 | 29900720 | 14.3G | Apple APFS |

# Verschlüsselter USB Stick (APFS Volume)

1. Installation von **apfs2hashcat** (umfasst das Kompilieren der Sourcen)
2. Hash extrahieren durch "Copy-and-Paste" aus dem Logfile/der Konsole.

```
$ sudo ./apfs-dump-quick \  
/dev/sda2 \    # /dev/sda2 ist die Ziel APFS Partition  
/tmp/log.txt
```

3. Hash angreifen

```
$ hashcat -m 18300 fv2.hashcat \  
/usr/share/wordlists/rockyou.txt
```

## **Passwortwiederherstellung mit Hashcat**

# Hashcat - Einführung

Hashcat ist - Stand 2023 - das Tool zum Wiederherstellen von Passwörtern.

Liest ein(e Liste von) Hash(es) ein und prüft, ob einer der angegebenen Passwortkandidaten nach dem Hashen mit einem gegebenen Hash übereinstimmt.

- unterstützt über 350 Hash-Typen (mit einigen automatischen Erkennungen)
- unterstützt mehrere Angriffsmodi, z.B.,
  - Wörterbuch (ggf. mit Regeln)
  - Masken
  - Kombinationen aus Wörterbüchern und Masken
  - <Lesen von Passwortkandidaten aus stdin>
- Open-Source
- Kann zum Generieren von neuen Kandidaten verwendet werden.
- ist CUDA/OpenCL basiert und **auf entsprechenden Grafikkarten extrem schnell.**

# Hashcat - relevante Parameter

Angriffsmodi:

`-a0` Angriff mit Wörterbuch  
(ggf. mit Regeln `-r`)

`-a1` Kombinationsangriff  
Angriff mit dem Kreuzprodukt  
zweier Wörterbücher.

`-a3` Brute-force Angriff

`-a6` Hybridangriff  
Wörterbuch und Maske

Brute-force - Eingebaute Zeichensätze:

`?l` = `abcdefghijklmnopqrstuvwxyz`

`?u` = `ABCDEFGHIJKLMNOPQRSTUVWXYZ`

`?d` = `0123456789`

`?s` = `!"$%&'()*+,-./:;<=>?@[]^_`{|}~`

`?a` = `?l?u?d?s`

Definition von bis zu 4 eigenen Zeichensätzen  
ist möglich.

# Hashcat - Ausgewählte Regeln

| Name        | Function | Description                                    | Input    | Output           |
|-------------|----------|--|----------|------------------|
| Nothing     | :        | Do nothing (passthrough)                       | p@ssW0rd | p@ssW0rd         |
| Lowercase   | l        | Lowercase all letters                          | p@ssW0rd | p@ssw0rd         |
| Uppercase   | u        | Uppercase all letters                          | p@ssW0rd | P@SSW0RD         |
| Capitalize  | c        | Capitalize the first letter and lower the rest | p@ssW0rd | P@ssw0rd         |
| Toggle Case | t        | Toggle the case of all characters in word.     | p@ssW0rd | P@SSw0RD         |
| Reverse     | r        | Reverse the entire word                        | p@ssW0rd | dR0Wss@p         |
| Duplicate   | d        | Duplicate entire word                          | p@ssW0rd | p@ssW0rdp@ssW0rd |
| Append      | \$X      | Append X to the end                            | p@ssW0rd | p@ssW0rdX        |
| Prepend     | ^X       | Prepend X at the beginning                     | p@ssW0rd | Xp@ssW0rd        |
| ...         | ...      | ...  | ...      | ...              |



# Szenario 1: eine Pin Angreifen

## Ausgangssituation


Gegeben sein ein mit SHA256 gehashter 5-stelliger Pin in der Datei: `5_digits_pin.sha256`.

Hashwert:

`79737ac46dad121166483e084a0727e5d6769fb47fa9b0b627eba4107e696078`

## Angriff mit Maske

```
hashcat -m 1400 5_digits_pin.sha256 -a3 "?d?d?d?d?d"
```

- m 1400:** Modus für einen einfachen SHA256 Hash.
- a3:** bezeichnet einen Maskenangriffe
- "?d?d?d?d?d":** Beschreibt die Maske. Hier 5 Ziffern ( *digits*).

## Szenario 2: Ein (hoffentlich) einfaches Loginpasswort angreifen

### Ausgangssituation

Ein mit SHA512crypt gehashtes Passwort in der Datei: `password.sha512crypt`.

### Angriff mit Wörterbuch

```
hashcat password.sha512crypt -a0 /usr/share/wordlists/rockyou.txt
```

**-a0:** bezeichnet einen Wörterbuchangriff.

**/usr/share/wordlists/rockyou.txt:**

Das zum Angriff verwendete Wörterbuch; der Pfad ist der Standardpfad zum Rockyou Wörterbuch in Kali Linux.

## Szenario 3: ein komplexeres Passwort angreifen

### Ausgangssituation

Ein mit MD5 gehashtes Passwort in der Datei: `password.md5`. Ein erster Angriff mit Rockyou war nicht erfolgreich.

### Angriff mit Wörterbuch und Regelsatz

```
hashcat -m 0 password.md5 \  
        -a0 /usr/share/wordlists/rockyou.txt \  
        -r /usr/share/hashcat/rules/best64.rule
```

**-a0:** bezeichnet einen Wörterbuchangriff.

**/usr/share/wordlists/rockyou.txt:**

Das zum Angriff verwendete Wörterbuch.

**-r /usr/share/hashcat/rules/best64.rule:**

Der zum Beugen der Passwortkandidaten verwendete Regelsatz.

Der Regelsatz best64 hat sich in einem Wettbewerb als „bester“ Regelsatz erwiesen.

## Szenario 4: ein Passwort mit Salt angreifen

### Ausgangssituation

Ein MD5 Hash ist gegeben: `c84b5c34c9ff7d3431018d795b5975e5`. Weiterhin ist bekannt, dass der verwendete *Salt* `SALT` ist.

### Angriff

1. Modus für MD5+Salt heraussuchen (`-m10`); ggf. Beispielhash ansehen, um zu verstehen, wie der Hash aufgebaut ist.
2. Erzeugen des Hashes für Hashcat:

```
echo -n "c84b5c34c9ff7d3431018d795b5975e5:SALT" > salted.md5.hash
```

3. Mit Hashcat angreifen:

```
hashcat -m10 salted.md5.hash -a3 '?a?a?a?a'
```

# Szenario 5: Kombination von Wörterbuch mit eigenem Regelsatz

## Ausgangssituation

Wir greifen einen sogenannten langsamen Hash an und können deswegen nur wenige Passworte gezielt testen.

Aufgrund von Social Engineering/Ermittlungen wissen wir, dass die Person häufig kurze Worte (max 4 Buchstaben nimmt) diese aber oft verdoppelt und häufig die Worte mit einem Großbuchstaben anfangen lässt.

## Angriff

1. Erstellen eines fokussierten Wörterbuchs: **candidates.txt**.
2. Erstellen des Regelsatzes: **case.rule**.
3. Angriff mit den erstellten Wörterbuch und dem Regelsatz.

# Szenario 5: Kombination von Wörterbuch mit eigenem Regelsatz

## Angriff

### 1. Generierung von `candidates.txt`

Um sicherzustellen, dass wir keine Duplikate testen, wandeln wir alle Worte in Kleinschreibung um und filtern entsprechende Duplikate. Die Beachtung aller Varianten in Hinblick auf die Groß- und Kleinschreibung wird durch die Regeln sichergestellt.

```
$ grep -Po "[a-zA-Z]{3,4}(?=[^a-zA-Z])" \  
    /usr/share/wordlists/rockyou.txt \  
| tr [:upper:] [:lower:] \  
| sort -u \  
> candidates.txt
```

70

## Zu Bedenken

Die gezeigte Operation löst die Ordnung in der Datei auf und sortiert diese alphabetisch. Dies ist aber häufig nicht gewünscht - insbesondere wenn der Leak nach Verwendungshäufigkeit sortiert ist!

# Szenario 5: Kombination von Wörterbuch mit eigenem Regelsatz

## Angriff

### 1. Erstellen des Regelsatzes: `case.rule`

Um sicherzugehen, dass wir alle Varianten abdecken, brauchen wir drei Regeln.

|    |  |
|----|--|
| cd | Erst Groß-Kleinschreibung anpassen und dann duplizieren. |
| dc | Erst duplizieren und dann Groß-Kleinschreibung anpassen. |
| d  | Einfach nur duplizieren.                                 |

### 2. Angriff mittels Hahcat

```
hashcat -m 1700 hash.sha125 candidates.txt -r case.rule
```

71

## Tips

Das beherrschen von regulären Ausdrücken ist bei der Passwortrekonstruktion sehr hilfreich.

Der folgende Ausdruck liefert zum Beispiel alle 4stelligen Worte aus Rockyou mit Hilfe eines Lookheads, dass längere Worte filtert.

```
$ grep -Po "[a-zA-Z]{3,4}(?=[^a-zA-Z])" \  
/usr/share/wordlists/rockyou.txt
```

Das Passwort **TreeTree** würde sich damit erfolgreich wiederherstellen lassen.

## Szenario 6: Kartesische Produkt von zwei Wörterbüchern

### Ausgangssituation

Aufgrund von Social Engineering/Ermittlungen wissen wir, dass die Person sehr gerne zwischen deutschen Großstädten pendelt. Nachdem andere Versuche nicht zum Erfolge geführt habe, wollen wir jetzt Passworte der Art: "BerlinHamburg" testen.

### Angriff

1. Erstellen eines fokussierten Wörterbuchs durch "Googeln" von großen Städten.
2. Angriff durch Kombination des Wörterbuchs mit sich selbst.

```
$ hashcat -m 1400 hash.sha256 -a 1 big_cities2.txt big_cities2.txt
```



# Szenario 7: Wörterbuch mit Maske

## Ausgangssituation

Es ist bekannt, dass die Passwörter der Gruppierung häufig mit vier Zahlen und zwei Sonderzeichen aus einer sehr kleinen Mengen von Sonderzeichen (\$!.) enden. Davor kommt ein Wort mit ca. 4-8 Stellen in den typischerweiser "liebe/love/luv" vorkommt.

## Angriff

1. Erstellen eines fokussierten Wörterbuchs: **candidates.txt**
2. Angriff mit passendem Maskenangriff

## Szenario 7: Wörterbuch mit Maske

### Angriff mit Hybridangriff

`candidates.txt` enthält alle Begriffe aus `rockyou`, die die Anforderung erfüllen:

```
$ grep -oE "[a-zA-Z]*[Ll]((uv)|(ove)|(iebe))[a-zA-Z]*" \  
    /usr/share/wordlists/rockyou.txt \  
| sort -u \  
> candidates.txt
```

Angriff mit Hashcat:

```
$ hashcat -m 1400 hash.sha256 candidates.txt \  
-a 6 -1 '$.!' '?d?d?d?d?1?1'
```

74

### Beispiel

In diesem Falle verwenden wir einen Hybridangriff, der eine Wordliste mit einer Maske kombiniert. Hier definieren wir unseren eigenen „Zeichensatz“ mit dem Parameter `-1 '$.!'` und referenzieren diesen in unserer Maske später mit `?1`.

Ein Beispielpasswort, dass wir mit dem Ansatz ermitteln könnten, wäre:

| SHA256   | Passwort    |
|--|-------------|
| b9cace43df57bc694498bf4d7434f45a<br>8466c4a924f608d54fd279d24b3dc937 | ILuvU2023!! |

## Szenario 8: Passwörter mit Muster

### Ausgangssituation

Wir möchten ein Wörterbuch erstellen mit „Wörtern“, die Buchstabenvervielfältigungen enthalten, aber nicht länger als 16 Zeichen sind. Zum Beispiel: "aaaaBBBBcccc" oder auch "AFFFFFFE". Weiterhin soll die Liste nach der Länge der gefundenen Einträge aufsteigend sortiert sein und Zeichen, die keine Buchstaben sind, einfach gelöscht werden.

### Lösung

Heraussuchen entsprechender Wörter aus rockyou mittels Linux Kommandozeilenwerkzeugen.

```
$ grep -E "([a-zA-Z])\1{3,}" /usr/share/wordlists/rockyou.txt  
| grep -E "^.{4,16}$"  
| sed -E 's/^[^a-zA-Z]//g'  
| sort -u  
| awk '{print length " " $1}'  
| sort -n  
| sed -E 's/^[0-9]+ //'
```

---

### Alternative Aufgabenstellung

Sortierung der finalen Liste nach der Häufigkeit der Muster, angefangen mit dem häufigsten Mustern.

# Szenario 9: Passwörter bestehend aus Fragmenten

## Ausgangssituation

- Einer gegebenen Liste können wir nur entnehmen, dass alle Passwörter zusammengesetzt sind aus den Fragmenten: **ab**, **mem**, **li** und **xy**.
- Darüber hinaus ist immer eine Zahl vorangestellt und am Ende kommt ein Punkt (.) oder ein Ausrufezeichen (!).
- Die Länge scheint zwischen 6 und 16 Zeichen zu sein und Fragmente können sich wiederholen.

Beispiel: **1ablixyxy.**

## Vorgehen

1. Erstellen eines Basiswörterbuchs (**base.txt**) mit den Fragmenten als Einträge.
2. Erstellen von Regeln für das Voranstellen und Anhängen der entsprechenden (Sonder)zeichen.
3. Aus Basiswörterbuch das finale Wörterbuch für den Angriff generieren.
4. Mit dem finalen Wörterbuch und entsprechenden Regeln angreifen.

# Szenario 9: Generierung von Wörterbüchern aus Fragmenten

## Lösung

Zu Generierung aller Kombinationen aus den Fragmenten verwenden wir den Princeprocessor. Der Princeprocessor ist sehr schnell und ermöglicht es in Fällen die Ausgabe direkt an Hashcat durchzureichen und das Zwischenwörterbuch nicht explizit speichern zu müssen.

## Angriff

```
$ princeprocessor --pw-min=6 --pw-max=16 base.txt \  
| hashcat -m 1400 hash.sha256 \  
-r number_prepend.rule \  
-r sc_append.rule
```

77

Aufbau von `number_prepend.rule`:

```
^0  
^1  
...  
^9
```

Aufbau von `sc_append.rule`:

```
$.  
$!
```

Mit dem obigen Ansatz könnte zum Beispiel das folgende Passwort ermittelt werden:

| SHA256  | Passwort  |
|---|-----------|
| 8b11f8e8d487266a791d6d723a3e380c 38f49679735a7f3395ace4302e83dd0e | 8abxylxy. |

In diesem Falle wäre es auch möglich gewesen nur einen Regelsatz zu erstellen mit den passenden Regeln (zum Beispiel: `^1$.`, `^1$!`, ...) der Aufwand wäre hier jedoch höher gewesen und hätte keinen Nutzen gehabt.

Im Allgemeinen ist jedoch bei der Verwendung des Kreuzproduktes von Regeln immer darauf zu achten, dass keine (oder zumindest keine relevante Anzahl von) Regeln dupliziert werden. Ein Beispiel wäre das Kreuzprodukt aus einem Regelsatz für das optionale Anhängen einer Ziffer mit sich selbst. Sei der Regelsatz:

```
:  
$1  
$2
```

und würde man diesen mit sich selber kombinieren, um alle Fälle des Anhängens von keiner, einer bzw. zwei Zahlen abzudecken, dann würden folgende Regeln entstehen:

```
::  
:$1  
:$2  
$1$1  
$1$2  
$2$1  
$2$2  
$1:  
$2:
```

Wie zu erkennen ist, führen zum Beispiel die Regeln **\$1:** und **:\$1** jeweils zum gleichen Ergebnis und wären deswegen nicht effektiv.

## Szenario 10: Hashcat als Werkzeug zur Wörterbuchgenerierung

**Ausgangssituation** Gegeben sein 3 Wörterbücher [1]: **base1.txt**, **base2.txt** und **base3.txt**. Gesucht ist ein Wörterbuch, dass alle Kombinationen aus den drei Wörterbüchern enthält und bei dem alle Teilworte immer mit Sonderzeichen (-) voneinander getrennt sind.

**Beispiel** Sei **base1.txt**: "Kuh", "Schwein"; **base2.txt**: "Haus", "Villa" und **base3.txt**: "Baum", "Busch". Dann wäre das gesuchte Wörterbuch: "Kuh-Haus-Baum", "Kuh-Haus-Busch", ..., "Schwein-Villa-Busch".

### Vorgehen

1. Erzeugen des Kreuzprodukts der ersten beiden Wörterbücher.

```
$ hashcat --stdout base1.txt base2.txt -j '$-' > base1-base2.txt
```

1. Erzeugen des finalen Wörterbuchs durch Bildung des Kreuzprodukts der Ergebnisse aus Schritt 1 mit dem dritten Wörterbuch.

```
$ hashcat --stdout base1-base2.txt base3.txt -j '$-' > final.txt
```

---

[1] Die selbe Vorgehensweise lässt sich auch anwenden, wenn man ein Wörterbuch mit sich selber kombinieren möchte. 78

Die Hashcat Utilities Bibliothek hat auch noch weitere Werkzeug zum Kombinieren von Wörterbüchern, die viele Fälle sehr effizient abdecken (auch den besprochenen). Jedoch ist es gerade in Fällen, in denen komplexere Regeln zur Anwendung kommen sollen, häufiger sinnvoller/nowendig direkt Hashcat im "stdout" Modus zu verwenden, um die Zwischenwörterbücher zu generieren.

# Passwörter angreifen - Zusammenfassung

- Passwörter können vielfach effizient angegriffen werden.
- (gute bis exzellente) Kenntnisse über die Zielpersonen sind häufig notwendig.
- Viele Werkzeuge sind verfügbar (siehe auch Hashcat Werkzeuge, Princeprocessor, John the Ripper, etc.)
- Kleine etablierte Kommandozeilenwerkzeuge (tr, greb, sed, awk, ...) oder selbstentwickelte Werkzeuge (zum Beispiel in Python) sind häufig ergänzend notwendig und führen oft schneller zum Ziel als die Suche nach **dem** Tool.
- Insbesondere wenn es um die semantische Anreicherung von Wörterbüchern geht, dann sind (bisher) keine etablierten Werkzeuge vorhanden.
- Häufig führen nur Kombinationen von etablierten und eigenen Werkzeugen zum gewünschten Ziel.



## MD5 Hash eines trivialen Passworts

```
81dc9bdb52d04dc20036dbd8313ed055
```

Hinweise: Das Passwort ist kurz, besteht nur aus Ziffern und ist sehr häufig.

## MD5 Hash eines einfachen Passworts

```
7c6a180b36896a0a8c02787eeafb0e4c
```

Hinweise: Das Passwort besteht aus Buchstaben gefolgt von Ziffern und ist sehr häufig.

Sie können Hashcat (<https://hashcat.net/hashcat/>) verwenden oder ein Bash-Skript schreiben oder eine kleine Lösung in einer Programmiersprache Ihrer Wahl entwickeln.

# Sichere Passwörter

- Nehmen Sie kein Passwort, dass 1:1 in einem Wörterbuch oder Verzeichnis vorkommt.
- Nehmen Sie keine Szenepasswörter (zum Beispiel: acab, 1312, 88, ...).
- Je länger desto besser, aber keine ganz einfachen Sätze.
- Wählen Sie ein Passwort, dass sie sich merken können. Kombinieren Sie z.B. Dinge aus Ihrem privaten Umfeld, die aber niemand direkt mit Ihnen in Verbindung bringen kann. (D.h. die Namen Ihrer Kinder, Haustiere, etc. sind keine gute Wahl, aber ggf. das Modell Ihres Fernsehers in Kombination mit einer PIN und dem Namen Ihres ersten Smartphones getrennt durch ein paar Sonderzeichen).

# Literaturverzeichnis

- [SePass] SePass: Semantic Password Guessing Using k-nn Similarity Search in Word Embeddings; Maximilian Hünemörder, Levin Schäfer, Nadine-Sarah Schüler, Michael Eichberg & Peer Kröger, ADMA 2022: Advanced Data Mining and Applications Springer LNAI, volume 13726
- [PCFG] S. Aggarwal, M. Weir, B. Glodek and B. Medeiros, "Password Cracking Using Probabilistic Context-Free Grammars," in 2009 30th IEEE Symposium on Security and Privacy (SP); doi: [10.1109/SP.2009.8](https://doi.org/10.1109/SP.2009.8)
- [NGPCFG] S.Houshmand, S. Aggarwal and R. Flood, "Next Gen PCFG Password Cracking," in IEEE Transactions on Information Forensics and Security, vol. 10, no. 8, pp. 1776-1791, Aug. 2015, doi: [10.1109/TIFS.2015.2428671](https://doi.org/10.1109/TIFS.2015.2428671).
- [PAofPCFG] Hranický, R., Lištiak, F., Mikuš, D., Ryšavý, O. (2019). On Practical Aspects of PCFG Password Cracking. In: Foley, S. (eds) Data and Applications Security and Privacy XXXIII. DBSec 2019. Lecture Notes in Computer Science(), vol 11559. Springer, Cham. [https://doi.org/10.1007/978-3-030-22479-0\\_3](https://doi.org/10.1007/978-3-030-22479-0_3)
- [PlandPCFG] Houshmand, S., Aggarwal, S. (2017). Using Personal Information in Targeted Grammar-Based Probabilistic Password Attacks. In: Peterson, G., Shenoi, S. (eds) Advances in Digital Forensics XIII. DigitalForensics 2017. IFIP Advances in Information and Communication Technology, vol 511. Springer, Cham. [https://doi.org/10.1007/978-3-319-67208-3\\_16](https://doi.org/10.1007/978-3-319-67208-3_16)