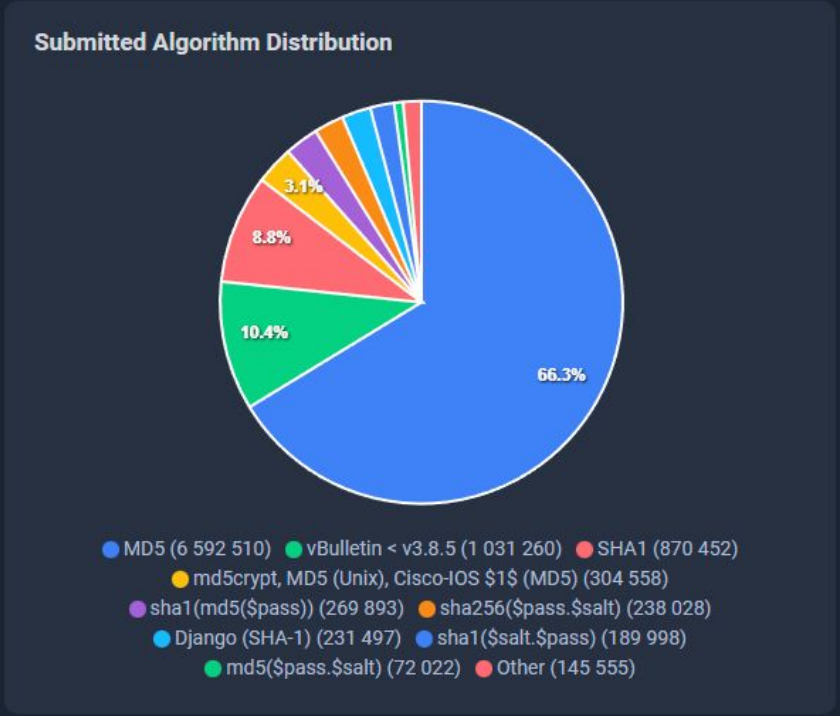
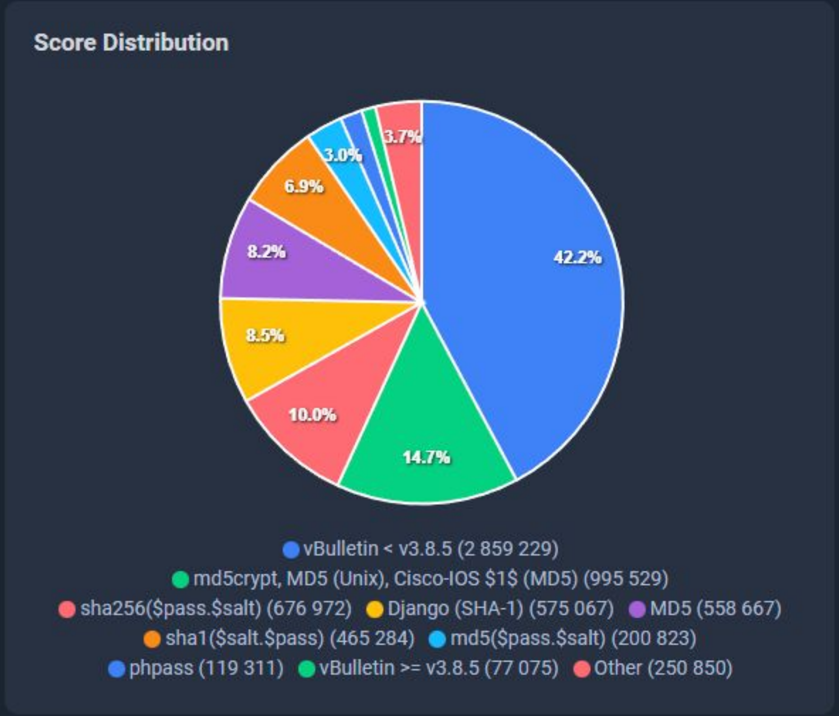
Leveling Up Password Attacks with Breach Data

Jake Wnuk

Support Security Research





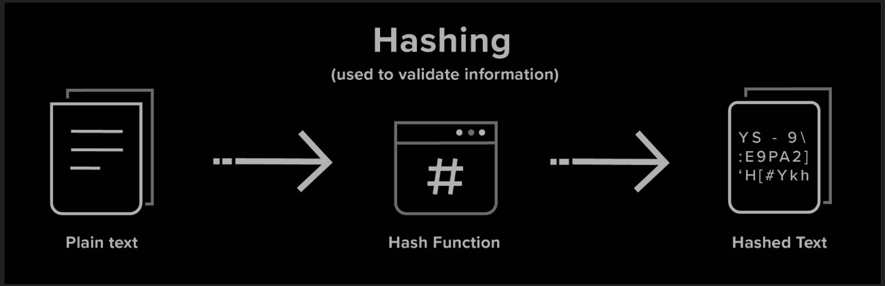
https://hashmob.net/

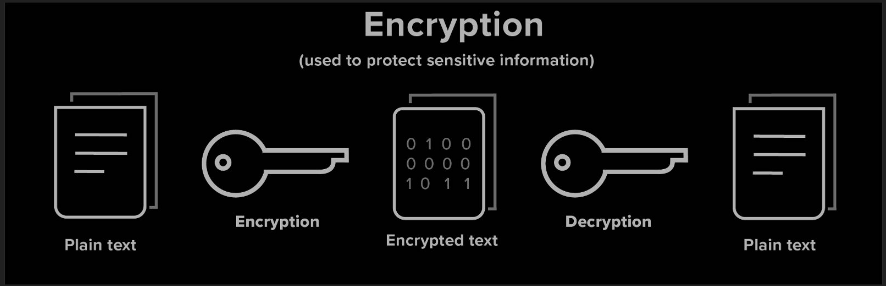
Topics:

An overview of our conversation today

- How are Secrets Passed In an Environment?
- Introduction to Hash Cracking
- Breaches & Password Data
- Data Driven Password Methodology
    - Optimizing Wordlists
    - Rule Raking & Ranking
    - Token Substitution Attacks
    - Token Swapping Attacks
    - Mutating Wordlists
- Application Example

How Are Secrets Passed In An Environment?





- Hashes are the result of a one-way algorithm to transform plaintext into ciphertext.
- Why do applications, environments, and systems use hashes?
  - Message and data integrity
  - Efficient computing and algorithms
  - Other cryptographic applications
  - Password storage and verification

Where Are Attackers Finding Hashes?

- Hashes are often used in security specific applications:
  - NTLM and Kerberos hashes, often within Active Directory environments
  - Application integrity and password management
  - Application authentication and other Identity and Access Management (IAM) Systems
- Hashes are often directly associated with system users or roles
- This makes hashes a lucrative target for security practitioners

Why Do Attackers Like Hashes?

- Hash cracking provides a consistent and reliable methodology for recovering secret material
  - Offline Attacks
  - Password Cracking
  - Secret Cracking
  - Intentionally Public Hashes
  - Note: Hashes do not always need to be “cracked” in order to be used

- How do attackers crack hashes?

Introduction to Hash Cracking

- This process can be replicated at millions of comparisons per second
- Attackers will use as much compute available to them to compromise hashes
  - With the advanced in cloud computing, this includes rented hardware
- Popular tools include Hashcat and “John The Ripper”

```
$ echo 'ThisIsMyPassword' | md5sum

75858444c091085e51a1f65718633b1f


$ echo 'ThisIsMyPassword1' | md5sum

0fbffba6ddf4511a724481e8f595b022


$ echo 'ThisIsMyPassword' | md5sum

75858444c091085e51a1f65718633b1f
```

Introduction to Hash Cracking

```
# simple usage
$ hashcat -m 0 -a0 hashlist wordlist -r rules


# common parameters
$ hashcat -m 0 -a0 hashlist wordlist -r rules --loopback --bitmap-max=24 -O


# other attack modes
$ hashcat -m 0 -a6 hashlist wordlist ?a
$ hashcat -m 0 -a7 hashlist ?1 wordlist -1 ?l?d
$ hashcat -m 0 -a3 hashlist password_masks
```

Breaches and Password Data

- What happens when hashes become public?
  - In 2009, the RockYou breach provided insight into how users were creating their passwords.
  - The breach exposed ~ 32 million plaintext passwords
- Why do security researchers love public hashes?
  - Real passwords provide the clearest insight into how users are setting secrets
  - Real passwords provide valuable data to improve password security
  - Real passwords are the gold standard when cracking passwords. Everything else can be abstracted.

Password Structure

- Password structure can be thought of in a few ways
  - The words and symbols that compose the password (tokens, n-grams/c-grams, lemma, ect)
  - The structure the password is in (mask)
- Password structure can be thought of in the form of masks which is a placeholder for a specific character in a password

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

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

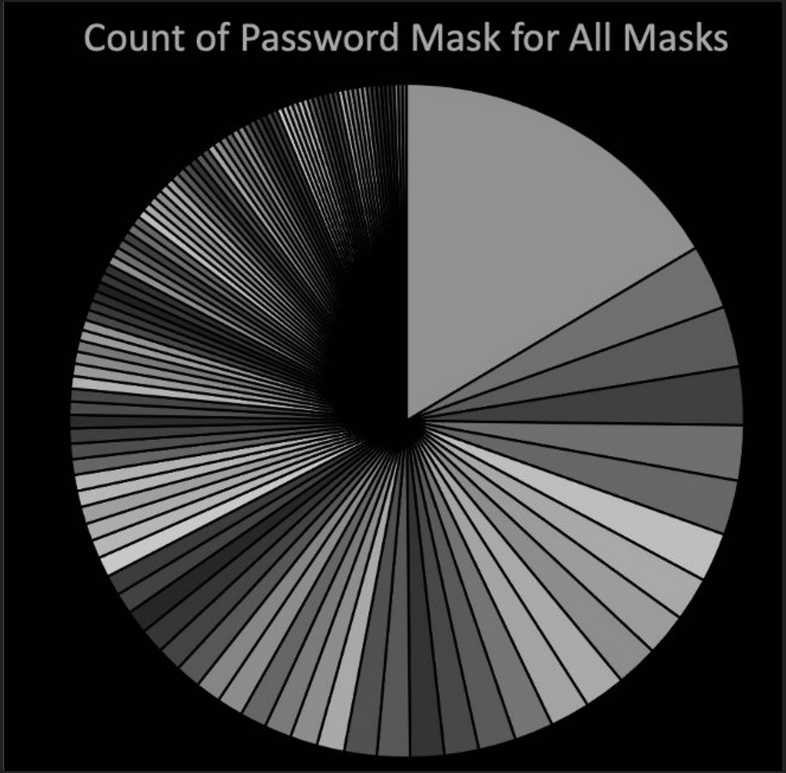
Password --> ?u?l?l?l?l?l?l?l
Pwd123 --> ?u?l?l?d?d?d

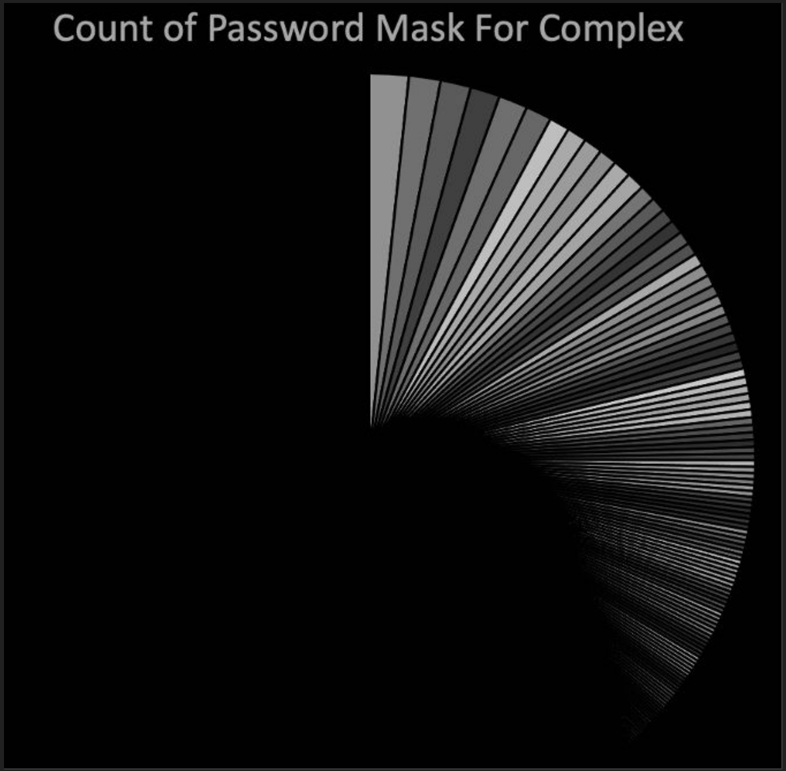
Password Structure

- Password structure can also be viewed from its components which we will refer to as tokens
- Humans tend to reuse secret material when setting passwords - names, locations, objects, religious terms, industry jargon, terminology, and other common themes are likely to reappear
- Tokens can be enumerated from popular passwords but are also highly specific to target sources
- It is not unusual for secret material to also be reused within environments, either directly or indirectly

```
123
love
100
me
my
password
the
ma
alex
big
1234
angel
friend
link
pass
123456
sam
john
jesu
19
red
12345
abc
babi
david
mike
jack
max
king
blue
daniel
admin
```

~ 1,000,000,000 Password Masks Analysis





Data Driven Attacks

- Optimizing Wordlists
- Rule Raking & Ranking
- Substitution Attacks
- Token Swapping Attacks
- Mutating Wordlists

Bash Aliases for Data Wrangling

```
# unique sort file
usort() {
    if [[ $# -ne 1 ]]; then
        echo 'unique sort file
inplace'
        echo 'EXAMPLE: usort
<FILE>'
    else
        LC_ALL=C sort -u $1 -T ./
-o $1
    fi
}
```

```
# get most common items in file
mode() {
    if [[ $# -ne 1 ]]; then
        echo 'find the most common
item in file'
        echo 'EXAMPLE: mode <FILE>'
    else
        LC_ALL=C sort -T ./ $1 |
uniq -c | LC_ALL=C sort -T ./ -rn
    fi
}
```

Introducing Maskcat

- Maskcat is a utility tool for Hashcat masks and password cracking (`cat` mask)
  - Makes Hashcat masks from stdin
  - Matches words from stdin to masks from a file argument
  - Substitutes tokens in wordlists using masks
  - Mutates stdin using masks to create new candidates
- https://github.com/JakeWnuk/maskcat
- Written in Go

```
$ echo 'ThisISaT3ST123!' | maskcat
?u?l?l?l?u?u?l?u?d?u?u?d?d?d?s:15:4
:333

$ cat masks.txt

?u?l?l?l?u?u?l?u?d?u?u?d?d?d?s

$ echo 'ThisISaT3ST123!' | maskcat
match masks.txt

ThisISaT3ST123!
```

Optimizing Wordlists

- An immediate benefit to storing breach data is the availability of password candidates
- Thanks to our mask analysis, we known that many people tend to set passwords in similar patterns
- If we took just the top 5,000 masks, we would cover around 86.8% of the plaintext passwords

```
$ head cracked_passwords.txt |
maskcat > cracked.lst

$ mode cracked.lst
   8 ?u?l?l?l?l?l?l?l?d?d?d?d?s
   7 ?u?l?l?l?l?l?l?s?d?d?d?d
   6 ?u?l?l?l?l?l?d?d?d?d?s

   5 ?u?l?l?l?l?l?l?d?d?d?d
   4 ?u?l?l?l?l?l?l?s?d?d?d
   4 ?u?l?l?l?l?l?l?l?s?d?d?d?d
   4 ?u?l?l?l?l?l?l?l?s?d?d?d
   4 ?u?l?l?l?l?l?l?l?l?d?s
   4 ?u?l?l?l?l?l?l?d?d
   4 ?u?l?l?l?l?l?d?d
```

Optimizing Wordlists

```
# making a wordlist
$ cat wordlist.lst | maskcat match top-5k-masks.txt > top5kmaskswords.lst
$ cat dumped-passwords.lst | maskcat match top-5k-masks.txt > top5kmaskswords-3.lst

# making a complex wordlist
$ cat wordlist1.lst | maskcat match top-5k-3to4ge8-masks.txt > top5kmaskswords-3to4ge8.lst

$ cat dumped-passwords.lst | maskcat match top-5k-3to4ge8-masks.txt >
top5kmaskswords-3to4ge8.lst

# sample of a list
$ head top5kmaskswords-3to4ge8.lst
    $01april
    $01august
    $01August
    $01autumn
```

Optimizing Wordlists

- no-ending: no special filtering, just the top x masks
- c8: filtered for complexity and length greater than or equal to eight (8)
- c8l: same as c8 but everything in lowercase
- nd: same as no-ending but skipping masks that are 100% digits
- leftovers: list containing all of the non-matched items

Wordlist	Size	Line Count
top5kmasks.lst	11GB	1,030,877,000
top15masks.lst	2.9GB	307,605,705
top5masks.lst	1.6GB	175,981,061
top5kmasks-c8.lst	1.3GB	123,148,699
top15masks-c8.lst	229MB	23,745,089
top5masks-c8.lst	120MB	12,849,424
top5kmasks-c8l.lst	1.3GB	123,148,699
top15masks-c8l.lst	229MB	23,745,089
top5masks-c8l.lst	120MB	12,849,424
top22masks-nd.lst	3.4GB	366,420,774
leftovers.lst	3.4GB	210,183,312

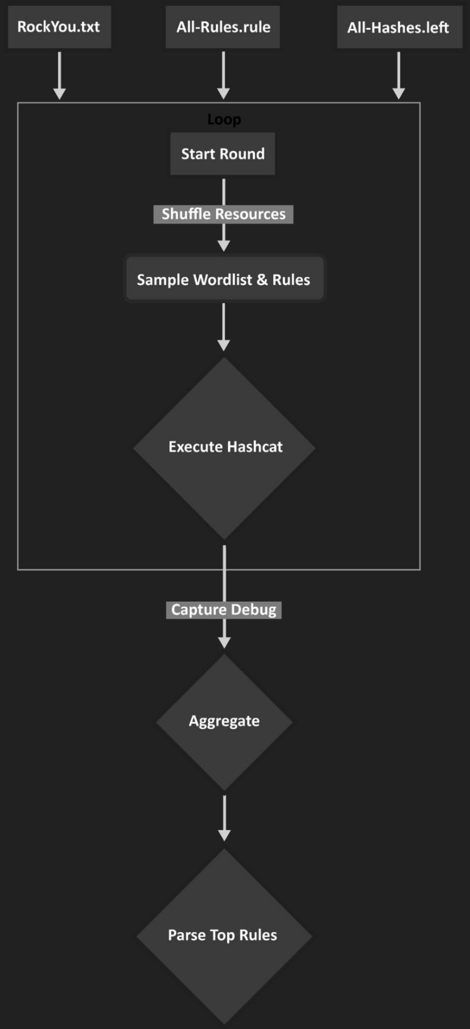
Optimizing Wordlists

Wordlist	Cracked	Cracked %	Keyspace
rockyou2021 (news ref)	2121189	40.799	8459060239
hibpv6	1597316	30.723	892631604
hashes.org-2019	1548285	29.78	522172105
top22masks-nd.lst	1545697	29.73	366420774
top15masks.lst	1181174	22.719	307605705
top5kmasks-c8l.lst	888222	17.084	123148699
top5kmasks-c8.lst	746127	14.351	123148699
top15masks-c8l.lst	463822	8.921	23745089
rockyou	420944	8.097	14344359
top5masks.lst	338345	6.508	175981061
ignis-1M	288167	5.543	1000000
leftovers.lst	279328	5.373	210183312
top5masks-c8l.lst	277234	5.332	12849424
SkullSecurityComp	275411	5.297	6693327
top15masks-c8.lst	201255	3.871	23745089
top5masks-c8.lst	109147	2.099	12849424

Rule Raking & Ranking

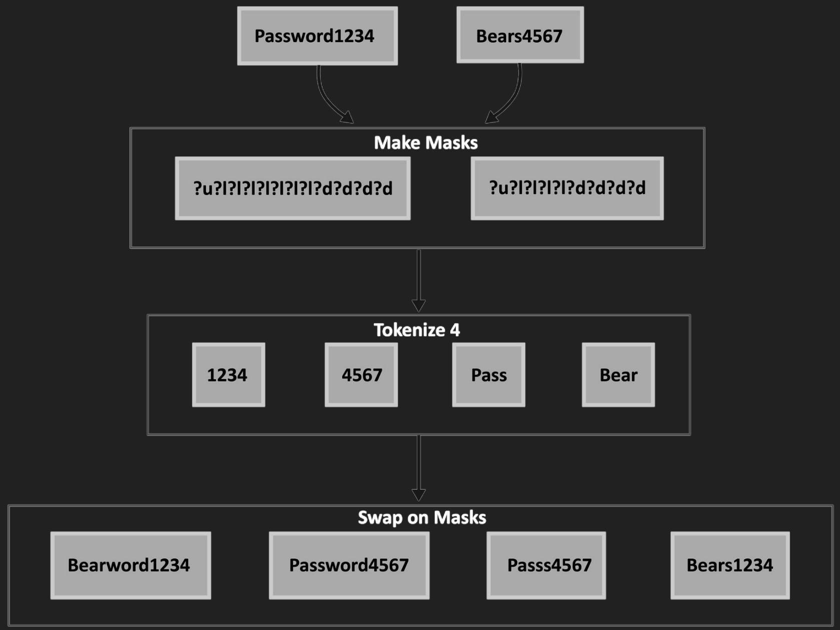
- Rule "raking" is the ability to generate new rules
- Rule "ranking" is optimizing existing rules using password data
- By maintaining a large password store with quality candidates, additional tests can be done to find high performing rules on specific subsets

```
$ hashcat -a 0 -m 1000 hash wordlist --debug-mode=1
--debug-file=debug.txt --potfile-disable --quiet -r rules.rule -O -o
/dev/null

$ cat ./attempts/*.txt > all.txt

$ mode all.txt
```



Introduction to Token Attacks

- If the best wordlists come from real passwords and passwords from the same source share many patterns, then the most logical step is to incorporate other founds from the same/similar source into new wordlists.
- The logic we want to replicate is:
  - Split plaintext into segments
  - Convert each token into a hashcat mask
  - Find other plaintext that contain the same mask
  - Swap the token into other plaintexts using the mask
- This provides two benefits:
  - Plaintext material is preserved
  - Plaintext masks/structure is preserved



Token Substitution Attack

- Secret material is often reused within an environment or user set.
- Likely tokens can be gathered from Open Source Intelligence (OSINT) or already known secrets
- We can attack probable secrets using token swapping to insert likely tokens into popular password masks.

```
$ cat topXmasks-passwords.txt
Winter2023
Summer2022
TheSummer1
MyPsswrd2!

$ cat tokens.txt
Bsides

$ cat topXmasks-passwords.txt |
maskcat sub tokens.txt
Bsides2023
Bsides2022
TheBsides1
MyBsides2!
```

Token Swapping Attacks

- When attacking passwords, candidates can quickly dry up or fall out of line with actual user patterns. We can use token swapping to preserve both structure and secret material to generate new candidates from existing founds.
- This process is highly non-deterministic and ideal for high entropy workflows with targeted attacks. This can turn very few candidates into a massive amount.

```
$ cat found.lst | shuf | maskcat mutate 6 >> mutate.lst

$ cat found.lst | shuf | maskcat mutate 5 >> mutate.lst

$ cat found.lst | shuf | maskcat mutate 4 >> mutate.lst

$ for i in {1..100}; do cat found.lst | shuf | maskcat mutate 8 >> mutate.lst; done;

$ usort mutate.lst
```

Token Swapping Testing

Case 1:

- Recovered........: 991/994 (99.70%)

Case 2:

- No swap found rerun:
  - Recovered........: 161/111762 (0.14%)
- Token swap:
  - Recovered........: 2386/111762 (2.13%)
- ~ 80 founds/hr compared to ~ 367 founds/hr

Case 3:

- Recovered........: 68038/8900340 (0.76%)

```
$ mode maskcat-masks.tmp | head
  3325 ?d?d?d?d?d?d?d?d?d?d:10:1:100
   3039 ?u?l?l?l?l?s?d?d?d:10:4:203
   2712 ?u?l?l?l?s?d?d?d?d:9:4:177
   2242 ?l?l?l?l?d?d?d?d:8:2:144
   2161 ?l?l?l?l?l?d?d?d?d:9:2:170
   2067 ?u?l?l?l?l?l?s?d?d?d?d:11:4:229
   1845 ?l?l?l?l?l?l?d?d?d?d:10:2:196
   1771 ?l?l?l?l?l?l?l?l:8:1:208
   1738 ?l?l?l?l?l?l?l?l?l:9:1:234
   1460 ?u?l?l?l?l?s?d?d?d:9:4:193
   1272 ?l?l?l?l?l?l?l?l?l?l:10:1:260
   1239 ?u?l?l?l?l?l?s?d?d?d:10:4:219
   1104 ?l?l?l?l?s?d?d?d?d:9:3:177
   1091 ?l?l?l?l?l?l?d?d:8:2:176
---
$ mode maskcat-masks.tmp | head
    107 ?u?u?d?u?u?u?u?u:8:2:192
    103 ?u?u?u?d?u?u?u?u:8:2:192
    103 ?l?l?l?l?l?d?l?l:8:2:192
    103 ?d?u?u?u?u?u?u?u:8:2:192
    100 ?u?d?u?u?u?u?u?u:8:2:192
     98 ?l?l?l?l?l?l?d?l:8:2:192
     73 ?l?l?l?l?l?l?l?d:8:2:192
     72 ?l?l?l?l?d?l?l?l:8:2:192
     42 ?l?l?l?l?l?l?d?u:8:3:192
     32 ?u?u?d?l?l?l?l?l:8:3:192
```

Wordlist Mutation

Sometimes when cracking, it can be all too easy to become over focused on found patterns and unfound patterns can fly under the radar.

This can also happen when not a lot of secret material is revealed, and offensive teams are “in the dark” about the nature of the hashes. When this happens, existing wordlists can be mutated against each other or with high performing wordlists to generate a significant number of likely candidates.

```
$ for i in {1..100}; do cat rockyou.txt| shuf |
maskcat mutate 8 >> mutate.lst; done;

$ for i in {1..100}; do cat rockyou.txt founds.lst
top5masks.lst | shuf | maskcat mutate 8 >>
mutate.lst; done;

$ usort mutate.lst

---

$ cat rockyou.txt | shuf | maskcat mutate 8 >>
rockyou-mutate.lst

$ cat rockyou.txt | shuf | maskcat mutate 7 >>
rockyou-mutate.lst

$ cat rockyou.txt | shuf | maskcat mutate 6 >>
rockyou-mutate.lst

$ cat rockyou.txt | shuf | maskcat mutate 5 >>
rockyou-mutate.lst
```

Application Example

Generate Candidates

```
# Run normal process then re-process founds

$ cat founds.lst | maskcat mutate 8 >> mutate.lst

$ usort mutate.lst


# Optional loop. Ranges from 4-12 normally.


$ for i in {1..100}; do cat founds.lst | shuf | maskcat mutate 8 >> mutate.lst; done;


# Can also include likely lists


  $ cat founds.lst rockyou.txt | shuf | maskcat mutate 8 >> mutate.lst
```

Filtering Down Candidates

```
$ cat mutate.lst | maskcat match target.hcmask > mask-mutate.lst
$ cat potfile | awk -F ':' '{print $NF}' | maskcat > f.tmp
$ mode f.tmp | head
  86472 ?l?l?l?d?d?d?d?d?d:9:2:138
  65678 ?l?l?d?d?d?d?d?d:8:2:112
  33214 ?d?d?d?d?d?d?l?l:8:2:112
  15691 ?d?d?d?d?d?d?l?l?l:9:2:138
  14038 ?l?l?l?d?d?d?d:7:2:118
  12515 ?l?l?l?l?d?d?d?d?d?d:10:2:164
   8938 ?l?l?d?d?d?d?d?d?d?d:10:2:132
   8065 ?l?l?l?d?d?d?d?d?d?d?d:11:2:158
   7049 ?l?l?l?l?l?l?d?d?d?d:10:2:196
   6159 ?d?d?d?d?d?d?l:7:2:86

# Can also use rli.bin to create multiple wordlists from the top masks to target
patterns
$ cat mutate.lst | maskcat match 5target.hcmask > top5mask-mutate.lst
$ cat mutate.lst | maskcat match 10target.hcmask > top10mask-mutate.lst
$ rli.bin top10mask-mutate.lst rli-top10mask-mutate.lst top5mask-mutate.lst
```

“Found” Rounds

```
# Another great part about candidate generation is that you can repeatedly use it
to find and enumerate patterns. After attacks, you can rotate potfiles or examine
founds for regeneration to focus on enumerating specific patterns:

$ cat potfile | awk -F ':' '{print $NF}' > founds1.lst

$ cat founds1.lst | maskcat > masks.tmp

$ mode masks.tmp | head

$ cat founds1.lst | shuf | maskcat mutate 5 > mutate2.lst

$ cat mutate2.lst | maskcat match topmasks.tmp > foundround.lst
```

Introducing OpenHashAPI

- How do you manage data storage?
- OpenHashAPI is an open source project that connects an SQL database to a REST API in order to store, manage, and query stored hashes.
- OpenHashAPI comes with a client and server component written in Go.
- OHA is designed for containerized deployment and focuses on storing password candidates.

```
$ docker run --restart always -it
--net=host --env-file config.env
openhashapi-server

$ docker run --rm -it --net=host
--volume ${PWD}:/data --env-file
~/.ohaclient.env jwnuk/ohaclient


$ ohaclient found 0 0.potfile

$ ohaclient search hashes.txt
```

https://jakewnuk.com

@JakeWnuk@infosec.exchange

https://github.com/jakewnuk

Thank you!

Tools:

- https://github.com/JakeWnuk/maskcat
- https://github.com/Scorpion-Security-Labs/OpenHashAPI-Server
- https://github.com/Scorpion-Security-Labs/OpenHashAPI-Client

More Content:

- https://jakewnuk.com/posts/optimizing-rules-w-entropy/
- https://jakewnuk.com/posts/optimizing-wordlists-w-masks/
- https://jakewnuk.com/posts/token-swapping-attack/
- https://jakewnuk.com/posts/advanced-maskcat-cracking-guide/