

# Computer Science 530 - Lab Assignment #2 -- Fall 2022

**Due: Friday September 23 2022, 4:30 p.m.**

In this lab you see first hand why passwords are such a weak approach to verifying identity, and why it is so important to choose strong passwords when stronger methods (such as second factors) are not available.

## Infrastructure for Lab

You will be using a Kali Linux appliance for this lab. The virtual appliance was created for last years lab and may be loaded into Virtual Box. It can also be loaded into VMWare if you prefer.

### Location of files

The ova file for this appliance is available in the CSci530 google drive in the folder for Lab 2. You can find the folder: [here](#)

Please note that you may need to login to google drive with your USC account in order to access these files.

### Some notes on this instance of Kali Linux

We have already loaded most of the programs you will need for this lab into the virtual appliance. When you start the virtual machine you will be asked to login, and the default account for the virtual machine is "student". Unfortunately, it seem that the account was set up, but the home directory was not created, and that makes it impossible to login initially to the "student" account. So, instead, for this lab, you will select "other" and then type in the account name "root". The Passwords for both both accounts is "c\$l@bLinuX". The third character is the letter "l" as in lab. Once you login as root, you might want to create the directory "/home/student" and then change the owner of the directory to "student" so that you may use that account in future lab exercises if needed. In any event, for this lab we would have made you "su" to root anyway for many of the operations, so the easiest course of action is to just remain logged in as root and complete the lab.

### Change the date on your virtual machine

In this lab we are intentionally using older versions of password hashing mechanisms because we want to labs to complete in a reasonable time, and our goal is to introduce you to the concepts of certain kinds of attacks. One of the programs you will be using would ask you to upgrade to a more recent algorithm if running as today's date. Therefore, for the purpose of this lab, you will need to set the system date in your virtual machine to Sepetember 1st, 2015. This can be done from your root shell using the command:

```
date 090112002015
```

## Overview

In this lab students will use a tool called "hashcat" to crack the passwords stored in a file. They were obtained from a Unix computer. Unix stores hashes of all its accounts' passwords in a single file. On older systems (pre-2000) the file was /etc/passwd; on newer ones it is /etc/shadow instead (/etc/passwd remains, but no longer holds the password hashes).

Passwords themselves, in clear text, are never stored. But gaining their hashes is a matter of copying the containing file. Given the file, an attacker can try at his leisure to figure out what the original passwords

were. That's what hashcat does. It has several techniques by which to crack:

## hashcat "attack modes"

straight [ = dictionary or wordlist attack ]

combination

toggle-case

brute-force [ implemented as subset of hashcat's so-called mask attack ]

permutation

table-lookup

You will use some of these in this lab. There are hashcat versions for linux, Windows, and OSX. You will run it within Kali Linux, a linux distribution in which hashcat comes pre-installed, against passwords produced and taken from a Fedora linux system. Our kali linux system is configured to use the md5 hash algorithm for processing passwords. Current linux distributions normally use sha512 by default. But ours has been set to md5 for

1. brevity, hence legibility, of output
2. speed of execution, considering the exercise is to be done in limited time

sha512 is preferred outside the tutorial context. (Incidentally the file that configures which hash algorithm is used for your system's passwords, on debian based systems like kali linux, is /etc/pam.d/common-password.)

The convention below is that where there is a command you should execute, it is shown in *italics*.

## Observing password format, production, and storage

Passwords get hashed coming and going. When originally created, the string supplied by the user is hashed and stored. When later used for login authentication, the candidate password supplied by the person who wants to log in is what gets hashed. That hash is compared for match against the stored one. But this is oversimplified a little. There are two wrinkles. First, a generated prefix called a salt is usually pasted in front of the user's password before hashing. Second, the processing algorithm differs from pure hashing. Though intimately based on one of the well-known hash algorithms, it does more to the password than just hash it. What ends up in /etc/shadow went through some additional steps.

Let's have the system create a password for a user the usual way, by using the passwd command, and see what it produces. Obtain a root shell (though su, or if you have been following directions, you are probably already logged in as root). This exercise calls on you to use certain files in /root/hashcat-exercise. Either the directory and files are already in place. Try:

*cd /root/hashcat-exercise*

Execute a script that employs the passwd command to produce an account password:

*./view-sample-password.sh*

Study its output. Note the stored/scrambled password, that is, what the system retains when you create a password. People often refer to it as the "password hash." But what is it the hash of, exactly? Not just the supplied plaintext password from the user, of course. That's what the salt is about, also included in the hash input. So we would expect it to be the hash of the salt-prefixed plaintext password. But that's visible in the screen output, and it isn't the stored version of the password. Try the same thing for yourself. Read from the screen what the random salt was, then manually key that in, followed by the password, and let the md5sum program hash all that:

*echo -n | md5sum*

(echo's -n option is necessary so as not to contaminate md5sum's input with a trailing newline.) What did you get? Is it the same as /etc/shadow's stored password? No? Is it even the same length?

For example the fixed length of md5 hashes is always 16 bytes. If represented by hexadecimal numerals (1 numeral represents half a byte-- "F" represents 1111, "9" represents 1001) it takes 32 of them to represent an md5 hash. But the length of the stored password your system produced when the script ran the passwd command is neither of these (look carefully at the screen). It's 22 bytes. So it can't be simply the hashed salted password. So what is it? It's the output of the crypt( ) function. Look briefly at a couple of man pages:

*man crypt*  
*man mkpasswd*

mkpasswd is a command that "encrypts the given password with the crypt(3) libc function using the given salt." So try:

*mkpasswd --method=md5 [whatever the password is] [whatever the salt is]*

(you would likely use instead "sha-256" or "sha-512" on your system, depending how it hashes passwords). This time you should have gotten the same thing as seen in /etc/shadow. That's because mkpasswd runs the same algorithm, through the crypt( ) function, as did the passwd command originally ("mkpasswd --help" tells you so). Here is an explanation describing [pure hashing vs crypt-based password processing](#).

## Now for the Actual Steps You should be performing (the lab Assignment 2)

### 1. Obtain needed password-bearing target files

You need some passwords to crack. We made a list of words to serve as passwords, some simpler some more complex. We assigned them to artificially named user accounts "crack01, crack02, ... , crack50" using the passwd command. We did it 3 times, configuring the system each time to employ a different hashing algorithm for password generation. We extracted the password (2nd) fields from the resultant 50 records in /etc/shadow. These are in files:

- crack-these-please-md5
- crack-these-please-sha256
- crack-these-please-sha512

All 3 files contain the generated password fields for the same set of 50 original plaintext passwords, but having processed them differently. You can find these files already in the /hashcat-exercise directory of your virtual machine.

Using these, let's get cracking.

### 2. Executing a dictionary attack

A dictionary attack uses a word database or wordlist. It tries all the words in the database against a given target password. The target password is presented in some scrambled form. The unix crypt scrambler produces one such form; there are others (Windows for example may scramble differently, and unix itself has "configurable scrambling" as we saw above, and it could be salted or not). The attack needs to know in advance exactly how the presented password scramble got scrambled. It processes the trial words from the list in that same way, looking for a match. (The test whether candidate password matches target password is whether candidate scramble equals target scramble.) hashcat has a "Straight" attack mode, which performs dictionary attacks. First you need a dictionary or wordlist. The wordlist has no fancy format, just text with a word on each line. There are lots of wordlists on the internet for download. Kali linux supplies several, in /usr/share/wordlists/. The well-known list rockyou.txt contains 14 million words, not particularly English but character combinations. By comparison the base vocabulary sufficient for everyday English fluency is in the low thousands of words. An educated native speaker's vocabulary numbers in the low ten-thousands.

A dictionary attack will crack all passwords that are in the dictionary, and none that are not. To see how it works, please make a wordlist. Among the 50 passwords in the "crack these please" set are

"hello", "wtaddtsbtk", and "dog". Please compose a 3-line text file with each of those on a separate line. Call it test-dictionary. You can do so with an editor, or instead this also works:

```
echo hello > test-dictionary
echo wtaddtsbtk >> test-dictionary
echo dog >> test-dictionary
```

Then apply it to the md5-scrambled version of the passwords:

```
hashcat -m 500 -a 0 crack-these-please-md5 test-dictionary
```

The 3 passwords are cracked, as seen on the screen. (Cracked passwords are also stored in a file called hashcat.pot, and you can use the -o option to direct hashcat to deposit the results in a file of your choosing.) Suppose we want to do the same thing, against the sha512-scrambled version. Try:

```
hashcat -m 500 -a 0 crack-these-please-sha512 test-dictionary
```

It doesn't work. It doesn't even try. The "-m 500" option tells hashcat to scramble the dictionary's words using md5. But it must apply to the dictionary words the same scrambling method originally applied to the passwords themselves, to have any hope of producing the same result and identifying any passwords. hashcat immediately disqualifies the given stored passwords because it sees their form isn't that of a password produced using md5. The stored passwords don't match the algorithm. So change the algorithm to match the stored passwords, from md5 to sha512. Change the "-m 500" to "-m 1800" which is hashcat's code for sha512:

```
hashcat -m 1800 -a 0 crack-these-please-sha512 test-dictionary
```

Now, again, the 3 passwords get successfully cracked. Do the same with a bigger dictionary:

```
hashcat -m 500 -a 0 crack-these-please-md5 500_passwords.txt
hashcat -m 1800 -a 0 crack-these-please-sha512 500_passwords.txt
```

This time none of our 3 passwords get cracked, but 7 different ones do. That's because the new dictionary lacks our 3 but contains these 7. If your dictionary contains all your passwords, cracking them all is time consuming:

```
hashcat -m 500 -a 0 crack-these-please-md5 50-crack-these-please
```

Note that using sha512 takes noticeably more time than md5.

As a principle, using a hash algorithm that requires more processing imposes a burden upon the password attack. Some password implementations employ hash algorithms incorporating deliberately slowed key derivation functions (e.g., bcrypt, PBKDF2). Their objective is to reduce the feasibility of cracking. If you use one to produce your passwords, you are ipso facto less vulnerable to cracking compared with the usual general-purpose hash algorithm implementations. linux distributions don't generally offer these; OpenBSD does. Curiously, one of the design goals of a "good" hash algorithm for its use as a signature producer is speed. For that purpose, speed's a virtue. Paradoxically, speed makes the algorithm "bad" for its use as a password scrambler since it facilitates cracking. Like many features in computer science evolution, the adoption of fast general-purpose hash algorithms for password scrambling antedated the current age of intensive cracking as an industry. There are current efforts to improve password hashing, and moves toward non-password and multi-factor authentication in many organizations.

hashcat comes with several side-utilities to ease your cracking life. Some are in /usr/share/hashcat-utils. Another, useful for targeted dictionary attacks, is maskprocessor. You can use it to make your own wordlists.

```
maskprocessor --help
```

You compose a mask like `?d` to represent any digit, or `?u?u` to represent any two uppercase letters. Try:

```
maskprocessor ?d?d |fmt -w 33
maskprocessor ?u?u |fmt -w 84
maskprocessor --custom-charset1=ABCDEF?d ?1?1 | sort |fmt -w 50 [ note: the ?1?1 are not
lowercase letter Ls but numerals for the number one ]
```

These generate, respectively, all the 2-digit numbers, all the 2-letter words, and all the 2-digit hex numbers. With imagination, you can construct wordlists containing patterns you consider likely to appear among the passwords subject to your attack, accelerating it.

### 3. Executing a brute force attack

A brute force attack tries the entire keyspace. As such it undertakes a very large volume of work. Comparatively, a dictionary attack is fast but quite possibly ineffective. When it can't find a password it lets you know quickly. It finds what it looks for, doesn't find what it doesn't, and exits. Whereas, a brute force attack is fully effective but slow. It will find the password for you, maybe within your lifetime. For our tutorial purposes we can't afford something that takes a lifetime, or even a half hour. So, artificially, we will work with mostly short passwords and use the md5 algorithm, which is relatively fast to operate compared to sha512:

```
openssl speed md5
openssl speed sha512
```

These show that, in the same amount of time, md5 processes more data than sha512.

One of the attack modes defined in hashcat syntax is "Brute-force." But its documentation deprecates it, "This attack is outdated. The Mask-Attack fully replaces it." Mask attack's documentation emphasizes that it encompasses brute force:

"Advantage over Brute-Force The reason ... is that we want to reduce the password candidate keyspace to a more efficient one....

"Disadvantage compared to Brute-Force There is none.... Even in mask attack we can configure our mask to use exactly the same keyspace as the Brute-Force attack does. The thing is just that this cannot work vice versa."

So mask attack has brute force covered, and supplants it. (It is invoked as the "-a 3" attack mode which however they still call "Brute-force," not "mask," in the documentation.) Try: `hashcat -m 500 -a 3 crack-these-please-md5 ?l?l [ lowercase letter L here ]`

If you wanted a traditional brute-force attack you could use mask `?a?a`, because `?a` covers all the 95 characters a US keyboard can produce. `?a` is unrestrictive, whereas `?l` is limited to just 26 of those:

```
hashcat -m 500 -a 3 crack-these-please-md5 ?a?a
```

This too will get the same "ww" but will take a little longer, because it traverses the larger keyspace for any/all 2-letter words (all 9025 of them). Notice hashcat prints these numbers for you in its screen output "Progress.: 9025/9025 (100.00%)".

In general you should try to reduce the password candidate keyspace to a more efficient one, like `?l?l` in this case. It proved a winning bet. However, had we chosen `?u?u` it would have been efficient but a loser, since our set of actual passwords didn't have any two-character uppercase ones.

Running a crack with larger alphabet sizes and, especially, larger mask/password lengths can add up fast and overwhelm your test. The time difference between the above 2 commands that deal with 2-character words is short enough we don't much notice. But extending it to 3-character passwords (`?l?l?l` and `?a?a?a`) the difference turns material for purposes of a tutorial exercise. On my system, instead of finding all the 3-character lowercase alpha passwords in 22 seconds, hashcat finds all the 3-character

ones of any keystroke value in 18 minutes 9 seconds (we have 6 of them). That's 50 times as long. That extra 3rd character generates 95 times as many password candidates to potentially try, so it could take 95 times as long to run worst case. The longest password in the set is 12 characters. A brute force attack would get all of them, but take several quadrillion times longer than our 3-character affair. Brute force on these with our consumer equipment is beyond reach.

#### 4. Executing a mask attack

The brute force attack is a mask attack, special case.. But let's apply the mask attack for its normal purpose, which is "to reduce the password candidate key space to a more efficient one." That means a smaller one, but not just any smaller one, rather a smaller one populated by high-probability candidates (thus weeding away low-probability ones not worth the bother). An example with our password set might be to crack those with 4 characters that start and end with a vowel. Our password set contains one of those ("into"). Here we can introduce hashcat's custom character set feature. It allows defining an arbitrary subset of the characters, then requiring a character in a mask to fall within that subset. The vowels, for example, are a subset of interest.

```
hashcat -m 500 -a 3 crack-these-please-md5 --custom-charset1=aeiou ?1?l?l?1 [ numeral one - lowercase L - lowercase L - numeral one ]
```

The ?1?l?l?1 mask represents a pair of lowercase letters sandwiched between leading and trailing vowels. It tends to express what we want. It isn't exact, but it's good enough. It finds our 4-character password "into" that starts and ends with vowels.

5. Applying hashcat rules Given a dictionary, rules can be applied to it that modify its words into similar ones that people might choose for passwords. For example, people will commonly convert SanDiego to sANdIEGO, or SANDIEGO, or anDiegoS, ogeiDnaS, or SanDi3go, exemplifying respectively case toggling, capitalization, letter rotation, spelling reversal, and leetspeak substitutions. Then they choose one of these cool tricky ones for their password. In anticipation, a cracker could augment his dictionary with all these. But he would have to separately add the variants for every entry in the dictionary. Rules take a dictionary and do that programmatically and dynamically at crack time.

I found this clarifying illustration at [suprafortix.wordpress.com](https://suprafortix.wordpress.com)

\$1 says "add numeral 1 at the end of each word" and c says "capitalize the first letter of each word." So where the above wordlist does not contain password1 nor Password, used together with a rule file containing "\$1" and "c", in effect it does. Without the rule file you will not catch password1 but with it you will.

Let's emulate the diagram by making 3 files-- myhashes, mywords, myrules. myhashes will hold the hashed versions of several "tricky" passwords users might choose (like cALIFORNIA). mywords will contain their regular "untricky" versions (like California), as a dictionary. myrules will express the modifications that produce the tricky from the untricky (like "t" for toggle).

Run the following script (key in, or copy/paste) to produce a file of password hashes of the 6 word variants you can see in the script:

```
while read LINE
do
    mkpasswd --method=md5 $LINE
done<<EOF > myhashes
cALIFORNIA
esrever
kroc
h3llo
d0g
j0hn
EOF
```

Now make a dictionary file of these common words from which the variants were derived. Its contents are:

**California**  
**reverse**  
**rock**  
**hello**  
**dog**  
**john**

Name your file "mywords". The third element is the file containing the rules which express the variations. Make a file containing:

**t**  
**r**  
**}**  
**se3**  
**so0**

Name your file "myrules". Referring to hashcat's rules for rules, you see that "t" says to "Toggle the case of all characters in word", "r" says to "Reverse the entire word, "}"Rotates the word right," "se3" replaces all instances of e with 3, and "so0" replaces all instances of o (lowercase o) with 0 (numeral zero). That's how you get from California to cALIFORNIA, from reverse to esrever, from rock to kroc, and so forth. These are generated on the fly, then tried. (Incidentally, therefore speed counts. So we are told, "The rule-based attack is like a programming language designed for password candidate generation.... [But ] Why not stick to regular expressions? Why re-invent the wheel? Simple answer: regular expressions are too slow.") Run your crack:

*hashcat -m 500 -a 0 -r myrules myhashes mywords*

Note that a rule crack tests for the variants it has generated but not the originals. If California as well as cALIFORNIA is present among the passwords you are testing, you won't catch the former with this test. Simply run hashcat twice, telling it what you want it to do both times (omitting "-r myrules" the second time, in order to test for the originals). Consider in which order to run them, depending on your assessment of the words' probability of use (do you have sly users or just regular guys?).

## What to turn in - The Assignment itself

The assignment:

1. Imagine a "file-based/static/pre-computed" type dictionary attack for cracking all the words in a language of 100,000 words. Suppose, when utilized as passwords, these are hashed as-is (unsalted), and the result then stored. ("Hashed" here means processed as the passwd command does.) To attack this, the cracker creates his dictionary in advance by 1) hashing all 100,000 words from first to last, then 2) re-sorting his dictionary on the hashes. Then, given a hashed password to crack, he simply looks it up and there he finds the original, plaintext password.

Without salt:

- a) the number of different ways a password can come out if hashed using no salt, is \_\_\_\_\_.
- b) the number of entries there will be in the dictionary the cracker must create is \_\_\_\_\_.

Now imagine that a 2-byte salt is introduced, randomly chosen then prefixed to each word before it is hashed and stored for use as a password.

With salt:

- c) the number of different ways a password could come out if hashed when prefixed with a random 2-byte salt is \_\_\_\_\_.

- d) the number of entries will there be in the dictionary the cracker must create is \_\_\_\_\_.
- e) if all the words in the language are 8 characters long and resolve to hashes 86 bytes long, thus requiring 94 bytes to store each mapped pair (dictionary entry), then the number of gigabytes the cracker's dictionary must occupy is \_\_\_\_\_.

2. Use the Mandylion "Brute Force Attack Estimator" Excel spreadsheet ([a slightly modified version](#)). Suppose you want a password that requires the rest of your life for a PC to crack. You have 50 years to live. How many days (live each to the fullest) is that? In the spreadsheet, consider passwords consisting of numerals ("Numbers") only.

- a) the length of the numbers-only password that requires at least 50 years to crack, according to the spreadsheet, is \_\_\_\_\_ characters?
- b) account for Moore's law. It says computing power doubles every 2 years. The spreadsheet is dated. It reflects the computing power of 10 years ago. For today, you need to increase its computing power assumptions by a factor of 32 (having doubled 5 times over the 10 years). Do so by entering 32 as the "Special factor" in cell G1 (which is applied in the "computing power" cell, E24, as a multiplier). Thus, with *today's* computing power, the length of the numerals-only password that requires at least the rest of your life to crack is \_\_\_\_\_ characters.
- c) account for Moore's law's continued operation. Let's assume Moore's law doesn't stop. (There's debate about that. But let's set it aside because if Moore's law's potential to continue raising cracking power is blunted, GPU advances or specialized cracking silicon may more than fill the gap.) Then today's isn't the right computing power for the upcoming 50 years' calculations. I say that on average (less near term, more far term) the upcoming power is 2.5 million times today's (approximately). Using 2.5 million as your future computing power, the length of the password that requires at least 50 years to crack becomes \_\_\_\_\_ characters. (Multiply the current special factor by yet a further  $2.5 \times 10^6$ )
- d) if you then made the one change of allowing mixed random characters (spreadsheet's "PURELY Random Combo of Alpha/Numeric/Special") instead of confining your password to numerals only you should be able to use a shorter password with equal effect. The shortest "mixed character" password that'll last 50 years is \_\_\_\_\_ characters.