COMP2700 Lab 3

Identification and Authentication

In this lab we will look at the theoretical aspects of password-based and biometric-based authentication (part I), and the practical aspects of password cracking (part II). At the conclusion of this lab, students should be able to:

- evaluate a password scheme security against brute force attack,
- distinguishes between different uses of biometric for authentication,
- perform simple practical password cracking exercises using a state-of-the-art password cracking tool.

For this lab, you are asked to complete Exercise 1 to Exercise 8.

After the lab, there will be a lab quiz (Lab 3 Quiz). The lab quiz will test your theoretical knowledge as well as practical knowledge. The Lab 3 quiz will include a password cracking challenge, so make sure you do the exercises below to get familiar with hashcat. More details will be provided in the Wattle page for the lab.

Part I. Password-based authentication, password counting, entropy and biometrics

Exercise 1

Consider an alternative scheme of password authentication between a claimant and a verifier depicted in the figure below, where h stands for a one-way hash function.

Notice that during the verification process, the claimant has to enter the plaintext password, and hashing of the password is done by the claimant, instead of the verifier. What could be a potential security issue with this authentication scheme?

For this exercise, you can assume that the communication channel between the claimant and the verifier is secure, so the attacker cannot intercept the password hashes in transit. But you may assume that the attacker can somehow obtain the table containing the hashes from the verifier.

服务器端哈希：
更安全，因为即使攻击者获得了哈希表，仍需破解哈希才能获取明文密码。
服务器可以增加安全措施，如使用盐值（Salt），防止彩虹表攻击。
客户端哈希：
存在严重安全隐患，因为哈希值本身即为认证凭证。
攻击者一旦获得哈希表，可以直接使用哈希值进行认证，而无需破解密码。
改进建议：
始终在服务器端进行哈希运算，并确保使用加盐以增加哈希值的唯一性。
使用更强的认证协议（如Kerberos）来替代容易受到Pass-the-Hash攻击的NTLM。

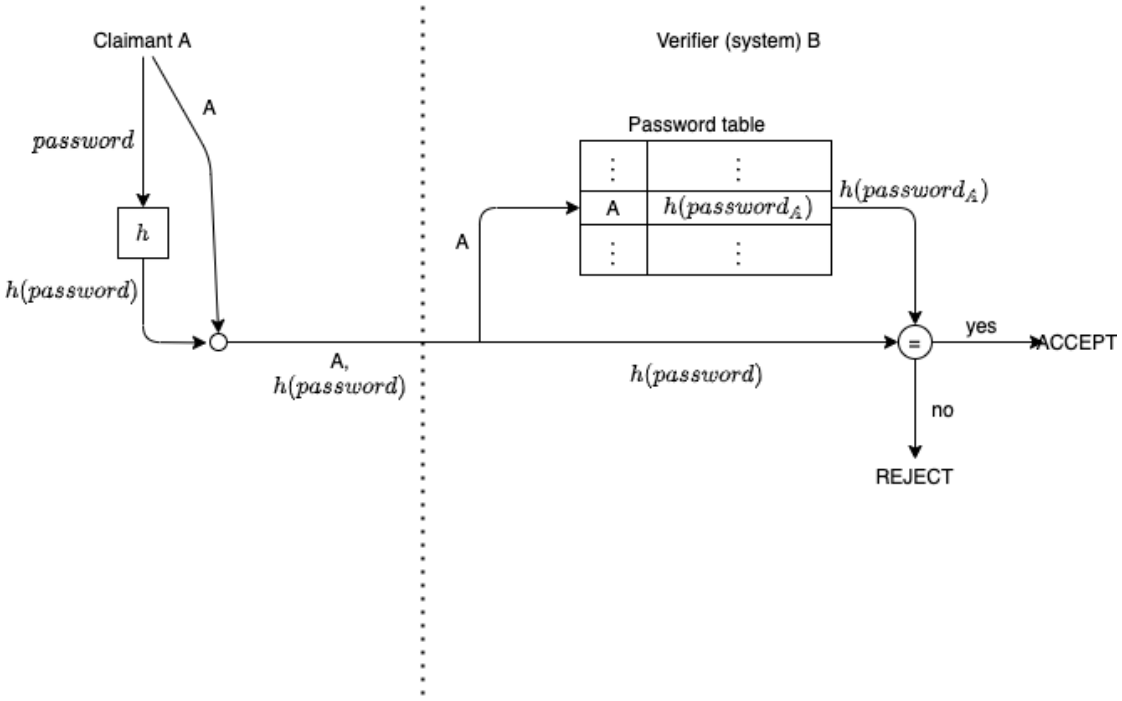

Figure 1: Diagram of a password authentication protocol.

Exercise 2

For each of the following password policies, calculate how many possible passwords satisfy the given policy.

1. A valid password is exactly 5 character long and each character is a lower case letter (a-z).
2. A valid password is at most 5 character long and each character is either a lower case letter (a-z) or an upper case letter (A-Z).
3. A valid password is exactly 10 character long, each character is either a lower case letter (a-z), an upper case letter (A-Z), or a numerical digit (0-9), and it must contain at least one numerical digit.

Exercise 3

Consider a fingerprint scanner with FMR=0.5% and FNMR=0.8%. For simplicity, let us assume that the Failure to Acquire (FTA) rate is 0%, so the device will capture fingerprint features perfectly at every scan.

Suppose this fingerprint scanner is used in an authentication server. To prevent a person from creating more than one account, the authentication server requires every new user registration to provide a sample of their fingerprints through the fingerprint scanner, which is then checked against the fingerprints of currently registered users; if a match if found, the registration is rejected.

To log in to the server, a user will provide their username and scan their fingers. The server will look up the stored fingerprint of the user and match it against the scanned prints – authentication succeeds if and only if they match.

Assume that we currently have 200 registered distinct individuals, each has their fingerprints stored in the registration system.

1. What is the probability that a registered user fails to authenticate to the server?
2. Suppose Alice is a new user who wants to create an account. What is the probability that the server rejects Alice's registration?
3. Suppose Bob is a registered user. Bob tries to create a new user account. What is the probability that Bob would succeed?

Exercise 4

In some Windows operating systems, to maintain backward compatibility, password hashes of local users in a computer are stored in two different form: the LM hash and the NT hash. The NT hash is computed directly from the user's password, whereas the LM hash is computed from the case-insensitive version, i.e., the user's password is converted to its uppercase version, prior to hashing. Let us assume that only the letters of the English alphabet are used in passwords.

1. Suppose the attacker manages to crack the LM hash of a password to obtain the non-case-sensitive version of the user's password. What is the upper bound of the number of attempts required by the attacker to guess the actual (case-sensitive) password? Assume the password length is n.
2. In general, for a password of length n, which one is the better strategy to brute force the password: brute force the LM hash first followed by guessing the case-sensitivity, or brute force directly the NT hash? Justify your answer by comparing the upper bound of the number of attempts required in both strategies.

Part II: Password cracking with hashcat

For this part, we will be doing some practical exercises in password cracking, using a well-known tool called hashcat. Hashcat has a very rich set of features and it is not possible to cover all of them in this lab, but hopefully it will provide you with enough basics to try the more advanced features yourself.

We will be using the lab VM that you have set up in Lab 1. You will need to log in as the user admin2700 for the following exercises.

Hashcat can use both GPU and CPU to crack passwords. By default, it will try to use GPUs, so if you are running it in the lab VM, using its default setup, you may encounter an error saying 'No devices found/left'. To surpress this error and to force hashcat to use CPU to crack the password, you need to add the option '–force' to every hashcat command you use.

For this exercise, you do not need to use the lab VM, if you have your own Linux installation. Hashcat should work in any Ubuntu installation so it is not specific to the lab VM.

Hashcat is a highly customisable password cracker. We will only look at a small subset of its capabilities here: dictionary attack, brute force attack (via patterns) and rule-based attack.

Downloading the lab files

For this lab, you are provided with some password hashes to crack. Download the file `md5_hashes.txt` using the following commands and save it to the home directory of admin2700.

```
$ cd ~/
$ wget http://users.cecs.anu.edu.au/~tiu/comp2700/md5_hashes.txt
```

The file `md5_hashes.txt` contains a list of password hashes generated using the MD5 hash function, one of the early hash functions for password storage, that is now considered insecure. We use this simple password hash function in the next few exercises to allow us to experiment with different password cracking strategies using the limited computation resource in the lab VM.

The attack methods we will see next are applicable to more complicated hash functions, such as linux crypt, but at a (much) reduced performance (we'll come back to this in Exercise 8 below).

Important notes on running hashcat in virtual machines

Hashcat is designed to take advantage of GPUs, and as the virtual machines used in this lab do not support GPU virtualisation, there may be some compatibility issues.

If you encounter errors such as 'illegal instruction' when running hashcat, you can try to use the legacy version, which is also installed in the lab VM (both the Azure Labs version and the VirtualBox version). Simply replace the command 'hashcat' with 'hashcat-legacy'.

For the purpose of cracking passwords, they should be interchangeable, but the information displayed in the status screen is slightly different.

Dictionary attack

This attack method, as the name suggests, uses a 'dictionary' (a list of commonly used passwords, obtained via, e.g., password leaks from various sources). For this lab, we will use a dictionary from `cracklib`, that is already installed int he lab VM, located in

```
/usr/share/dict/cracklib-small
```

The syntax for this attack method is as follows:

```
hashcat -a 0 -m 0 target_hashes dictionary_file -o output_file --force
```

We will go through the options one by one:

- The option `-a 0` specifies the attack modes, which is a 'straight' attack mode, that simply uses the dictionary file to crack the passwords. You can use

  ```
  bash
  hashcat --help
  ```

  to see all available options. For example, the help page shows the following options for attack modes.

  ```
  - [ Attack Modes ] -
    # | Mode
   ===+======
    0 | Straight
    1 | Combination
    3 | Brute-force
    6 | Hybrid Wordlist + Mask
    7 | Hybrid Mask + Wordlist
  ```

- The option `-m 0` specifies the type of hashes to crack. The number 0 here indicates that it is MD5 hash. MD5 is a type of hash function. Hashcat supports many different types of hash functions; see the help page for details.
- The file `target_hashes` contains the password hashes we want to crack. This needs to be replaced by the actual file you are targetting.
- The file `dictionary_file` is the file containing a list of words that will be used to crack the pasword hashes. Replace this with the actual dictionary file you will use.
- The option `-o output_file` specifies the name of the file containing the passwords that are successfully cracked.
- The option `--force`, as mentioned earlier, is used to force hashcat to use CPU to do the search. This is only needed in this lab because we are running hashcat in a VM. If you run this on an actual hardware with GPU, you will likely not need this option.

  Note: if you use `hashcat-legacy`, do not include the `--force` option.

Exercise 5

Use the dictionary attack to recover passwords in the hashes provided in `md5_hashes.txt`, using the cracklib dictionary in `/usr/share/dict/cracklib-small`. Which passwords did you manage to crack?

Hashcat outputs a number of statistics – one interesting number is the `Speed`, which gives the hash rate (the number of hashes generated per second). What is the hash rate that you observed in your attempt?

Note: Hashcat caches the hash-and-password combinations found so far in the file `/.hashcat/hashcat.potfile`, so if you run the same query repeatedly, the second and subsequent runs may terminate quickly and do not produce an output file (as it assumes the hashes were already cracked). If you want to restore your hashcat run to the initial state, simply remove the file `/.hashcat/hashcat.potfile`.

Brute Force Attacks

This attack mode simply tries all combination of characters to crack the hashes. This is only feasible for short passwords, that are not easily cracked using dictionary.

The syntax for this attack mode is as follows (assuming we're still cracking MD5 hashes):

```
bash
hashcat -a 3 -m 0 target_hashes pattern -o output_file --force
```

Notice that the option `-a` now uses `3` (instead of `0`), which indicates a brute force mode. The other new argument here is the `pattern` which specifies the pattern to brute force.

The brute force patterns refer to the character sets that hashcat uses in brute force attack. Hashcat has the following character sets defined (taken from the help page of hashcat):

```
- [ Built-in Charsets ] -

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

Each character set is referred to by prefixing it with a `?`. For example, to refer to the character set of all lower case letters, we use `?l`. Some character sets, such as `?a`, are defined by combining other character sets. For example, `?a` is obtained by combining `?l` (lower case), `?u` (upper case), `?d` (digits) and `?s` (symbols).

We can now specify patterns such as:

- `?l?l?l?l`: This refers to passwords of length 4 and each character is a lower case.
- `?u?l?l?l`: This refers to passwords of length 4, that starts with an upper case letter, followed by three lower case letters.

For example, the following command searches for passwords of 4 character long and each character is an upper case letter.

```
bash
hashcat -a 3 -m 0  target_hashes ?u?u?u?u -o output_file --force
```

You can also define a custom character set by combining built-in character sets. For example, the following command defines a custom character set `?1`, which is a combination of lower case letters and digits, using the option `-1 ?l?d`.

```
bash
hashcat -a 3 -m 0 target_hashes -1 ?l?d ?u?1?1?1 -o output_file --force
```

This command brute-forces a four character password where the first character an upper case letter, and each of the remaining three characters is either a lower case letter or a digit.

Exercise 6

Use the brute force attack mode to crack more password hashes in `md5_hashes.txt`. What new passwords did you recover? Note that he complexity of brute force is exponential in the length of the patterns, so you might want to restrict the length of the patterns to brute force, and limit the size of the character set you use for longer patterns.

Rule-based attacks

This is perhaps the most advanced method. The basic idea is to use an existing dictionary word to generate new password candidates, based on certain heuristics. The heuristics could be some well-known password transformations, e.g., adding digits to the end, or at the beginning of a word, substituting some letters with numbers, etc.

The syntax for this attack mode is the same as dictionary attack, but with an added option `-r`:

```
bash
hashcat -a 0 -m 0 target_hashes dictionary_file \
-r rule_file -o output_file --force
```

The option `-r rule_file` says that the words in the `dictionary_file` should be transformed according to the rules stated in the file `rule_file`.

Note: The backslash character \ (followed by a newline) at the end of a bash command tells bash that the command continues to the next line. This is useful if you want to break your (very long) command into two lines or more for readability. The above command can be typed as a single line (without the \) if you want.

Here are some examples of simple rules. For more details, see the hashcat wiki page (https://hashcat.net/wiki/doku.php?id=rule_based_attack).

Name	Function	Description	Example Rule	Input Word	Output Word
Nothing		Do nothing (passthrough)	:	p@ssW0rd	p@ssW0rd
Lowercase	l	Lowercase all letters	l	p@ssW0rd	p@ssw0rd
Uppercase	u	Uppercase all letters	u	p@ssW0rd	P@SSW0RD
Capitalize	c	Capitalize the first letter and lower the rest	c	p@ssW0rd	P@ssw0rd
Rotate Left	{	Rotate the word left.	{	p@ssW0rd	@ssW0rdp
Rotate Right	}	Rotate the word right	}	p@ssW0rd	dp@ssW0r
Append Character	$X	Append character X to end	$1	p@ssW0rd	p@ssW0rd1
Replace	sXY	Replace all instances of X with Y	ss$	p@ssW0rd	p@$$W0rd
Prepend Character	^X	Prepend character X to front	^1	p@ssW0rd	1p@ssW0rd

These functions can be combined to form more complex transformations. For example,

```
c ^2^1
```

will transform 'password' into '12Password', and

```
sa4 $1$2
```

will transform 'password' into 'p4ssword12'.

The rule file is simply a list of rules. Each rule in the rule file is applied to every word in the dictionary; so if you have N rules, then hashcat will generate N times the number of words in the dictionary.

Some examples of rule files can be found in the the directory /usr/share/hashcat/rules/. But try to solve the next exercise by writing your own rules, and only consult those example rules when you are stuck.

Exercise 7

Use the rule based attack to recover the remaining passwords from `md5_hashes.txt`. What rules did you use?

Exercise 8

To see the effect of a good password hashing algorithm, such as linux crypt, on attacks on passwords, we will now attempt to crack a password hash generated using the SHA512-based linux crypt algorithm. For this, we first generate a password hash for a very simple password ("hello") and save it to a file called `crypt_hash.txt`.

```
bash
$ openssl passwd -6 "hello" > crypt_hash.txt
```

This will generate a salted password hash. The salt is randomly generated, so you will likely get different hashes if you repeat the command. Now use hashcat to try to crack the hash in `crypt_hash.txt` you just generated:

```
bash
$ hashcat -a 0 -m 1800 crypt_hash.txt /usr/share/dict/cracklib-small \
-o crypt_out.txt -O --force
```

Here we use the option `-m 1800` which tells hashcat that the hash type is SHA512 crypt. We use `-O` to tell hashcat to use an `optimised` option to speed up the hash calculation (but at the cost of the reduced length of salt it can handle).

Compare the hash rate you see in this attempt vs. the hash rate you see in Exercise 5. How much slower is the hash rate for SHA512 crypt?

Note that you may see hashcat displaying the line:

```
bash
[s]tatus [p]ause [b]ypass [c]heckpoint [q]uit =>
```

This just means that hashcat thinks that the password cracking will take a while, so it's giving you an option to, e.g., check the current cracking status (`s`), or to quit (`q`). In this case you may simply wait it out as it should not take more than 3-4 minutes.

Further reading

The following tutorials

- https://laconicwolf.com/2018/09/29/hashcat-tutorial-the-basics-of-cracking-passwords-with-hashcat/
- https://laconicwolf.com/2019/03/29/hashcat-tutorial-rule-writing/

contain some more tips and tricks that you may find useful, especially the hybrid attacks. The hashcat wiki (https://hashcat.net/wiki/) contains a comprehensive list of resources for more advanced attacks.