Humans are bad at passwords.

It’s true. We’re terrible at making them, we’re terrible at remembering them, and we’re terrible at assessing their quality. It’s difficult to emphasize just how terrible we are without demonstrating how easy they are to break— so that’s exactly what we’re going to do.

Let’s take a look at perhaps the most powerful password cracking tool of our time: [hashcat](https://hashcat.net/hashcat/). Used by hackers and security professionals alike, hashcat serves as both a formidable enemy and a bleak reminder of just how terrible we humans are at handling security on our own.

Prerequisites

This is not an introduction to password cracking. That can be found [here](https://cryptokait.com/2019/09/24/everything-you-need-to-know-about-password-cracking-for-the-national-cyber-league-games/). Instead, this is a more advanced guide, therefore, we have some prerequisites.

* You should be able to recognize data formats such as hexadecimal and base64.
* You should have a basic familiarity with command-line tools. In particular, you should know how to pass flags and other arguments to command-line tools.
* Ideally, you should have a computer with a dedicated graphics processing unit (GPU) and up-to-date drivers. Hashcat works best with a dedicated GPU, but it isn’t strictly necessary for competitions such as [National Cyber League (NCL)](https://www.nationalcyberleague.org/). This also means that hashcat won’t perform well in most virtual machines, though there are exception.
* Examples in this blog target Windows 10, since it’s easiest to get a dedicated GPU working on Windows. If you’d rather use a different platform, you’ll need to have enough familiarity with both platforms to be able to translate commands. If you’re using a version of Windows prior to 10, some of the instructions in this blog won’t work for you. Typically, if you were to run these commands in Kali, for example, you would need to replace each occurrence of .\hashcat.exe with hashcat, and you may need to add additional flags to the command, such as -f.

Background

If you’re already familiar with the basics of encryption and the need for password hashing, you may want to skip this section.

What is a password hash?

Modern applications rarely store passwords in plaintext—that is, in an unencrypted form. Instead, they use a form of encryption called *hashing*. Passwords provided by users are first hashed before they’re stored in a database.

Hashing is a one-way cryptographic process. The same input will always result in the same output, but it’s not possible to go the other direction without a lot of work. When a user attempts to log in, the application will hash the attempted password and check whether it matches the stored hash. If they’re the same, the user has entered the correct password.

An ideal hashing algorithm produces output that appears random. It’s easy to get the hash if you know the password, but there’s no clear way to get the password from the hash. In order to determine the password, an attacker has to try to encrypt every possible password, comparing the resulting hash with the hash that they want to crack. They can improve their odds a bit by limiting their attempts to passwords that humans would likely use—e.g., dictionary words—but this still takes time, and it doesn’t guarantee success if a strong password is used. When we talk about *cracking a hash* or *cracking a password*, we’re usually referring to the process of automatically attempting a large number of passwords until we find one that matches the hash we have.

For a long time, these process was deemed sufficient. A hacker that compromised an application’s database was left with a list of hashes. They couldn’t easily obtain the passwords from the hashes, so they were left with useless gibberish.

Over time, people realized that this process alone was still insufficient for three reasons:

1. Humans are notoriously bad at coming up with random data, which also means we’re bad at coming up with our own passwords. This means a lot of people choose the same passwords, which means a lot of users have the same hash. We know roughly how common each popular password is—for example, the single most popular password is 123456. By matching plaintext password frequency with hash frequency, hackers can get a pretty good idea of which hashes correspond to which passwords. The fifth most popular password probably corresponds to the fifth most frequent hash, or something in that vicinity.
2. If half of the users in a database share the same hash, and a hacker manages to crack that hash, they will have the password for a lot of accounts. The number of hashes a hacker has to crack is a lot lower than the number of users.
3. Simple hashes are susceptible to rainbow table attacks. Someone can pre-compute the hashes for a large number of possible passwords. This takes a while, but once the tables are generated, they can easily be stored and shared. As long as the hashes are organized, an attacker can quickly look up each hash in the table to obtain the input password to which it corresponds.
4. Cracking 100 hashes usually doesn’t take much longer than cracking 10 hashes. Cracking software attempts each possible password, then compares the output hash to the list of target hashes. There’s no need to launch a separate attack for each hash.

To solve this, applications started using *salt*. Salt is randomly generated data that’s added to the user’s password prior to hashing, and it’s stored alongside the resulting hash in the database. Both the password and the salt serve as input to the hashing function. Two users might have the same password, but they should have different salts, so the hashes will be different. When the user tries to log in, the salt—which is saved as plaintext in the database—is included in the input to the hashing algorithm.

This solves all three problems:

1. Frequency comparisons won’t work because every user has a unique hash. Even though the attacker knows the salts, they don’t have enough information to determine whether any two hashes represent the same password.
2. Since there are (probably) no duplicate hashes, there are the same number of hashes as there are users. Cracking one hash won’t provide access to multiple accounts.
3. Rainbow attacks can’t reasonably be used because the salts are truly random. Instead of one hash for each password, there now needs to be one hash for each password multiplied by the number of possible salts. That’s a huge number; the resulting tables would take too long to compute and would be too large.
4. Each guess that cracking software attempts now has to be combined with each possible salt, and a unique hash generated for each password-salt pair. This means cracking 100 passwords takes about 10 times longer than cracking 10 passwords.

In NCL, you may see both standalone hashes and salted hashes. Although best practices dictate that salted hashes should be used, simple hashes are still used by some applications, often for compatibility reasons. Additionally, it’s often easier to practice cracking simple hashes, so they’re useful for educational purposes.

A note on word lists

Typically, passwords are cracked with the help of word lists. These are giant text files containing lists of possible passwords. In more advanced scenarios, a word list may contain common password roots, and the password cracking utility modifies them in some way—for example, by appending sequential numbers to each password.

Making good word lists is an art in its own right. We won’t be focusing on developing word lists in this article; instead, we’ll be using a popular beginner word list known as “rockyou”.

In 2009, [word spread that RockYou had been hacked](https://www.helpnetsecurity.com/2009/12/14/serious-sql-flaw-could-have-compromised-millions-of-rockyoucom-users/). This was notable because RockYou was storing their passwords in plaintext—that is, they weren’t hashing passwords. This meant that every user’s password was immediately visible to the attackers. In the years since, the resulting list of passwords has been one of the most analyzed data sets for studying passwords, and it’s often used as a learning aide. You’ll likely find passwords from RockYou used in many different security competitions, including NCL.

Password Cracking in NCL

Installing haschat

1. If you don’t already have it, install 7-zip, preferably 64-bit: <https://www.7-zip.org/>
2. Download the latest hashcat binaries: <https://hashcat.net/>
3. Right-click on the hashcat archive you downloaded and choose 7-Zip -> Extract Here. You should get a folder with the same name as the archive, e.g. hashcat-6.1.1.
4. Download the RockYou word list: <https://gitlab.com/kalilinux/packages/wordlists/raw/32566c1ee24113098772aa9ca1c1a4d5a04432fe/rockyou.txt.gz?inline=false>
5. Move rockyou.txt.gz into the hashcat folder.
6. Right-click on rockyou.txt.gz and choose 7-Zip -> Extract Here.
7. If you want to conserve storage space, you can now delete the compressed archives you downloaded, preserving the extracted files.
8. In an empty area within the hashcat folder, hold Shift while right-clicking. A context menu will appear; choose Open PowerShell window here. Keep this window open for later; we’re refer to it as the PowerShell prompt.
9. Using your favorite text editor, create a new text file in the hashcat folder named hashes.txt. Keep it empty for now; we’ll be adding content to it shortly.

Basic hashcat usage

In NCL, you’ll often have multiple hashes of the same type that you’ll want to crack—typically about five. When they’re not salted, it’s important to attempt them all at once. It takes roughly the same amount of time to crack 1 hash as it does to crack 100—when they’re not salted. To do this, add each hash to your hashes.txt file on a separate line. Make sure to trim any excess white space.

Let’s take a look at a series of hashes:

|  |  |
| --- | --- |
| 1  2  3 | b24aefc835df9ff09ef4dddc4f817737  1f23a6ea2da3425697d6446cf3402124  219a942ed67a2712dd6b0ac887c96f3d |

These are plain MD5 hashes. MD5 is an archaic hashing algorithm; it should never be used for securing sensitive data such as passwords, but, unfortunately, many applications still use it for that purpose. Because it’s quick and usually used without salt, it’s often used for educational purposes as well. Note that it’s exactly 32 hexadecimal digits with no extra data that could represent a salt: this is the signature characteristic of MD5.

Copy these hashes into your hashes.txt file and save it.

In the PowerShell prompt from earlier, type the following and press Enter. Pay extra careful attention to punctuation and whitespace.

|  |  |
| --- | --- |
| 1 | .\hashcat.exe -m 0 -a 0 hashes.txt rockyou.txt |

If everything worked, you’ll see a lot of technical output, maybe a few warnings, a bunch of metrics, and, somewhere in the middle:

|  |  |
| --- | --- |
| 1  2  3 | 1f23a6ea2da3425697d6446cf3402124:manchester123  b24aefc835df9ff09ef4dddc4f817737:heartbreaker07  219a942ed67a2712dd6b0ac887c96f3d:Estadosunidos18 |

Congratulations, you’ve cracked your first passwords! The portion on the left of each line is the hash, and the portion on the right is the corresponding password.

Let’s break down the command:

1. .\hashcat.exe: This is the path to the program that we’re running. In this case, we’re running hashcat.exe, which is located in the current folder (.\).
2. -m 0 tells hashcat that it’s going to be cracking MD5 passwords. Hashcat assigns each supported algorithm a number that it calls a “hash mode”; since MD5 is so common for testing and practice, it was assigned 0. You can get a full list of hash modes from [the hashcat website](https://hashcat.net/wiki/doku.php?id=example_hashes) or by running .\hashcat.exe -h
3. -a 0 tells hashcat that we’re just using a simple word list attack. There are other attack modes with varying degrees of complexity; you can get a full list by running .\hashcat.exe -h
4. hashes.txt is the file that contains the hashes to be cracked, one per line.
5. rockyou.txt is an attack-mode-specific parameter. Each attack mode typically takes one or two additional parameters that are specified after the hash file. Since we’re using a basic word list attack, we specify one additional parameter: the word list file.

Let’s try this again with some different hashes:

|  |  |
| --- | --- |
| 1  2  3 | 4e49638765211345b24821f51d17ad3cb9edd624  f17881a3334e0cde99bc94fc9e561db26c8dbef7  5fe0e6ba117a363af16dae5e435f7c7d463ffa93 |

Notice that these hashes are a bit longer than the previous ones; they’re each 40 hexadecimal digits instead of 32. If you take a look at [hashcat’s list of example hashes](https://hashcat.net/wiki/doku.php?id=example_hashes), you’ll see that these match the format of SHA1. You’ll also see that the hash mode for SHA1 is 100.

Replaces the contents of your hashes.txt file with the new hashes, save it, and run the following command in the PowerShell prompt:

|  |  |
| --- | --- |
| 1 | .\hashcat.exe -m 100 -a 0 hashes.txt rockyou.txt |

You should find the same passwords as last time:

|  |  |
| --- | --- |
| 1  2  3 | f17881a3334e0cde99bc94fc9e561db26c8dbef7:manchester123  4e49638765211345b24821f51d17ad3cb9edd624:heartbreaker07  5fe0e6ba117a363af16dae5e435f7c7d463ffa93:Estadosunidos18 |

Like MD5, SHA1 is outdated and shouldn’t be used for hashing passwords, but it’s still in use in many applications. Notice that the only change in our command was the hash mode; we switched from 0 (MD5) to 100 (SHA1).

Let’s try something a little different. So far, we’re been relying on word lists. However, what if we know that a password follows a specific format? In that case, word lists might not be ideal.

For example, NCL flags often take the form SKY-????-#### where each ? is an uppercase letter and each # is a number from 0-9. What if we know that our passwords are all SKY-TEST- followed by four digits? This may sound oddly specific, but it’s not unusual for newly provisioned users in an enterprise to be given passwords that follow a common pattern like that. This isn’t a great security practice, and you’ll soon see why.

Update your hashes.txt file with the following hashes:

|  |  |
| --- | --- |
| 1  2  3 | $2y$05$GAwb9NeYRburDXjp3UDz7u60tmAPPE4e1YbA5G0MCcuPuVaQMP9KK  $2y$05$4./ppbmiGYjg3XQqJQrQqe6jp3nhR.2W/fkncOVAUZd5M7vOTfYi2  $2y$05$6kYkZrD8gLvA8264S.jluOIGadDcqKZO4B6G379Qq9kRiYTxCt.Z6 |

We haven’t seen hashes like these before. These are in crypt form; they contain multiple pieces of information separated by dollar signs ($). Breaking down the components:

1. 2y indicates the hash type. In this case, 2 is the main hash family, and y specifies the subtype. If you’re interested, [Wikipedia has an explanation for the choice of 2y](https://en.wikipedia.org/wiki/Bcrypt#Versioning_history). For our purposes, we just need to figure out the algorithm and the corresponding hashcat hash mode. Searching [hashcat’s list of example hashes](https://hashcat.net/wiki/doku.php?id=example_hashes) for $2y$ doesn’t yield any results, but $2 brings us to a line that mentions $2\*$; the \* means “anything”, so we can assume that includes $2y$. You’ll see that it corresponds to bcrypt, a modern, secure hashing algorithm, and that the hash mode is 3200.
2. 05 indicates the cost factor. In simple terms, this determines how many times the password is run through the algorithm; the idea is that as computers get faster, the cost factor will be increased, allowing bcrypt to remain secure. Algorithms that are too fast are insecure because it’s easy for tools like hashcat to attempt a large number of passwords in a short time frame. As of 2019, a secure cost factor would be closer to 10, but we’re going to practice with 5 to keep the hardware requirements reasonable.
3. The base-64-ish string that follows is the salt followed by the hash. We don’t need to worry about separating them; hashcat is capable of parsing crypt-formatted hashes for us.

In addition to dealing with a much more secure algorithm and salt, we’re also going to need a new attack method. Our word list attack isn’t going to work unless our word list just so happens to contain a lot of passwords beginning with SKY-TEST-. Let’s test this just to be sure:

|  |  |
| --- | --- |
| 1 | .\hashcat.exe -m 3200 -a 0 hashes.txt rockyou.txt |

No dice. We could wait for hashcat to try every password in rockyou.txt. In the name of science, let’s see how long that’d take. Press the S key on your keyboard to get a status update from hashcat:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | Session..........: hashcat  Status...........: Running  Hash.Type........: bcrypt $2\*$, Blowfish (Unix)  Hash.Target......: hashes.txt  Time.Started.....: Mon Dec 09 22:45:46 2019 (56 secs)  Time.Estimated...: Mon Dec 09 23:21:20 2019 (34 mins, 38 secs)  Guess.Base.......: File (rockyou.txt)  Guess.Queue......: 1/1 (100.00%)  Speed.#1.........:    20152 H/s (5.13ms) @ Accel:8 Loops:2 Thr:8 Vec:1  Recovered........: 0/3 (0.00%) Digests, 0/3 (0.00%) Salts  Progress.........: 1139712/43033155 (2.65%)  Rejected.........: 0/1139712 (0.00%)  Restore.Point....: 379904/14344385 (2.65%)  Restore.Sub.#1...: Salt:0 Amplifier:0-1 Iteration:22-24  Candidates.#1....: 03232005 -> xoxo94  Hardware.Mon.#1..: Temp: 54c Fan:  4% Util: 98% Core:1949MHz Mem:5005MHz Bus:16 |

At about 20,000 hashes per second (H/s), my GPU is going to take about 35 more minutes to attempt the remaining passwords in rockyou.txt. Depending on your hardware, your estimated time remaining might be far higher than that. I don’t know about you, but I’m impatient; press Q on your keyboard to quit hashcat.

We’re going to need to adjust our tactics a bit. Instead of attack mode 0—a straight dictionary attack—we’re going to switch to attack mode 3, which is a brute force attack. Our new command will look like this:

|  |  |
| --- | --- |
| 1 | .\hashcat.exe -m 3200 -a 3 hashes.txt 'SKY-TEST-?d?d?d?d' |

Let’s go over the new elements of this command:

1. -a 3: We’re using attack mode 3 now, which corresponds to a brute force attack. You can get a full list of attack modes and their IDs in the output from .\hashcat.exe -h; there are also more detailed descriptions on [hashcat’s website](https://hashcat.net/wiki/#core_attack_modes).
2. Attack mode 3 still only takes a single parameter, but it’s not a wordlist file this time. Instead, we have a *mask*. This tells hashcat the format of the passwords it should attempt. Let’s break this mask down:
   1. The single quotes surrounding the mask ensure that PowerShell doesn’t interpret any symbols has having a special meaning.
   2. SKY-TEST- is used verbatim in each password.
   3. Each occurrence of ?d will be replaced with a digit; this is called a charset. There are other charsets built into hashcat, and you can define your own. The built-in charsets include:
      * ?l: lowercase letters (a-z)
      * ?u: uppercase letters (A-Z)
      * ?d: decimal digits (0-9)
      * ?h: lowercase hexadecimal digits (0-9, a-f)
      * ?H: uppercase hexadecimal digits (0-9, A-F)
      * ?s: basic ASCII symbols (e.g., @)
      * ?a: printable ASCII characters (numbers, letters, symbols)
      * ?b: every possible byte value, from 0x00 to 0xff

With these parameters, hashcat will attempt every password from SKY-TEST-0000 through SKY-TEST-9999, though not necessarily in that order.

Run the command. Depending on your hardware, it may take a little longer than before, but eventually you’ll get the passwords:

|  |  |
| --- | --- |
| 1  2  3 | $2y$05$4./ppbmiGYjg3XQqJQrQqe6jp3nhR.2W/fkncOVAUZd5M7vOTfYi2:SKY-TEST-4990  $2y$05$GAwb9NeYRburDXjp3UDz7u60tmAPPE4e1YbA5G0MCcuPuVaQMP9KK:SKY-TEST-7365  $2y$05$6kYkZrD8gLvA8264S.jluOIGadDcqKZO4B6G379Qq9kRiYTxCt.Z6:SKY-TEST-1858 |

Even with a relatively secure algorithm, patterned passwords aren’t secure.

Beyond the basics

Once you’ve mastered the basics of hashcat, you’ll likely want to learn about more advanced features. Those are beyond the scope of this article, but they’re often required for harder NCL challenges. Here are a selection of links to essential resources on advanced topics:

* [hashcat wiki](https://hashcat.net/wiki/)
  + [Hybrid attakcs](https://hashcat.net/wiki/doku.php?id=hybrid_attack): this combines that two attack modes we practiced
  + [Rule-based attacks](https://hashcat.net/wiki/doku.php?id=rule_based_attack): a technique to perform much more complex mutations on password candidates beyond simple masks
  + [Extracting WPA and WPA2 hashes from PCAPs for use with hashcat](https://hashcat.net/wiki/doku.php?id=hccapx): hashcat is much, much faster than aircrack-ng for WPA and WPA2
* [ophcrack](https://ophcrack.sourceforge.io/): a more specialized tool specifically for cracking Windows hashes
* [John the Ripper](https://www.openwall.com/john/): an alternative to hashcat; useful for cracking algorithms for which hashcat lacks support, such as [some ZIP files](https://hashcat.net/forum/thread-5709.html)