What is Hashcat?

Hashcat is a powerful password-cracking tool that works by guessing passwords and checking if they match a given "hash" (an encrypted version of a password). It's often used for ethical hacking and penetration testing.

Task Overview:

1. **Objective**: Crack as many passwords as possible from a list of hashed passwords.
2. **Tool**: Use **Hashcat** to perform this task.

Steps to Use Hashcat:

1. **Understand the Input**:
    - You'll be given a file containing password hashes (these are encrypted versions of passwords). They look like random strings.
    - Example: 5f4dcc3b5aa765d61d8327deb882cf99 (hash for "password").
2. **Choose an Attack Mode**:
    - Hashcat has several ways to guess passwords:
        - **Dictionary Attack**: Tries passwords from a list of common passwords.
        - **Brute Force**: Tries every possible combination (time-consuming).
        - **Hybrid Attack**: Combines both methods.
3. **Identify the Hash Type**:
    - Password hashes can be in different formats like MD5, SHA-1, or bcrypt.
    - Use a tool like hashid or hashcat --example-hashes to identify the hash type.
4. **Run the Command**:
    - Example command:

hashcat -m 0 -a 0 hash_file.txt wordlist.txt

        - -m 0: Indicates the hash type (e.g., MD5).
        - -a 0: Attack mode (dictionary attack).
        - hash_file.txt: File containing the hashes.
        - wordlist.txt: File containing possible passwords.
5. **Analyze the Results**:
    - Hashcat will output the cracked passwords if it finds any matches.

When you create a password, the computer doesn't store it as plain words like "apple123" because that's too easy for bad guys to steal. Instead, it uses a **magic machine** (called a hash function) that turns your password into a secret code.

These secret codes look like this:

- For **MD5**, your password "apple123" might turn into: 5f4dcc3b5aa765d61d8327deb882cf99
- For **SHA-1**, it might turn into: 40bd001563085fc35165329ea1ff5c5ecbdbbeef

- For **bcrypt**, it might look like this:
  $2b$12$3CpLf5u4w2qYxG9xNxyI/OBC8xvw

What's happening?

These are **different types of magic machines** that do the same thing:
They scramble your password into a secret code, but each machine scrambles it in a different way. Some are older and less secure (like MD5), while others are newer and harder to crack (like bcrypt).

Why do we care?

If you're trying to crack a password, you need to know **which machine scrambled it** so you can use the right tools to unscramble it.

Think of it like solving a puzzle:

- If the code was made with the MD5 machine, you need MD5 tools to figure it out.
- If it was made with bcrypt, you need bcrypt tools.

When it comes to password security, **bcrypt** is the most secure and least hackable of the three formats mentioned. Here's why:

1. MD5 (Least Secure):

- **MD5** is an older hashing method that is very fast to compute.
- **Problem**: Since it's fast, hackers can try billions of password guesses in a short time, making it very easy to crack using modern hardware.
- **Verdict**: **Not recommended** for security.

2. SHA-1:

- **SHA-1** is more secure than MD5 but is also outdated.
- **Problem**: SHA-1 was considered secure for a long time, but researchers found ways to break it, especially with enough computing power. It is now considered weak and vulnerable to attacks.
- **Verdict**: **Not recommended** for high security anymore.

3. bcrypt (Most Secure):

- **bcrypt** is designed to be slow, meaning it takes more time to compute the hash.
- **Why it's good**: The slowness makes it harder for hackers to try many guesses quickly. Also, it has a feature called "salting," which adds random data to the password before hashing, making it even harder to crack, even if two people have the same password.
- **Verdict**: **Best choice** for security.

What is Hashing?

Hashing is a way to **transform** a password into a scrambled version (called a hash) so that the original password is not stored directly. This is done to keep passwords secure.

What the Question Means:

- **"Determine what type of hashing aggregate was used"** means you need to identify the **type of hash function** that was used to create the hash (the scrambled password).

In simpler terms:
You need to figure out **which "magic machine" or method** scrambled the password. There are many different machines (like MD5, SHA-1, bcrypt), and each one creates a different type of hash.

How to Find the Hash Type:

Each type of hash has a **distinct pattern**. For example:

- **MD5** hashes are 32 characters long and typically only use numbers and lowercase letters.
- **SHA-1** hashes are 40 characters long and also use numbers and letters.
- **bcrypt** hashes are much longer and contain $2b$, a version number, followed by a salt (random data).

By looking at the length or structure of the hash, you can figure out which type of hash function was used.

Example:

If you see something like:

- 5f4dcc3b5aa765d61d8327deb882cf99
    - It's likely **MD5**.
- 40bd001563085fc35165329ea1ff5c5ecbdbbeef
    - It's likely **SHA-1**.
- $2b$12$3CpLf5u4w2qYxG9xNxyI/OBC8xvw
    - It's likely **bcrypt**.

In short:

This question is testing if you can identify the **hashing method** used by looking at the **format** of the scrambled password.

What Does "Level of Protection" Mean?

When a password is hashed, the **level of protection** refers to how difficult it is for a hacker to **reverse** the hash and find the original password.

Factors Affecting the Level of Protection:

1. **Hashing Algorithm**:
    - Some algorithms are stronger and harder to crack than others. For example:
        - **bcrypt** is very secure because it’s slow, making it harder to crack using brute-force attacks.
        - **MD5** is weak because it’s fast and easy to crack with modern computing power.
    - So, the **type of hash function** directly affects the level of protection.
2. **Salting**:
    - **Salting** is when random data is added to the password before hashing. This makes the hash unique, even if two users have the same password.
    - If a password is **salted**, it adds an extra layer of protection because it makes it harder for attackers to use precomputed lists of hashes (called rainbow tables) to crack the password.

3. **Iteration Count**:
    - Some algorithms, like **bcrypt**, allow you to control how many times the hashing process is repeated (iterations). The more iterations, the longer it takes to crack the password, making it more secure.

What This Question Wants:

You need to assess how **resistant** the password protection mechanism is to attacks like:

- **Brute-force attacks** (where hackers try every possible password combination).
- **Dictionary attacks** (where hackers use common passwords or wordlists to guess passwords).
- **Rainbow table attacks** (where hackers use precomputed tables to crack hashes).

Example:

- **bcrypt** offers **high protection** because it is slow, uses salting, and can have a high iteration count.
- **MD5** offers **low protection** because it is fast and doesn't use salting, making it easy for hackers to crack.

In short:

The question is asking you to determine **how hard or easy it is to crack the password** based on the hashing mechanism being used.

The Problem:

If a **password database** gets exposed to hackers, they can use the hashed passwords to attempt cracking them and gaining access to users' accounts. The goal is to **make this process much more difficult** or nearly impossible, even if the hashes are exposed.

What the Question is Asking for:

You need to think about **security controls** that could be added to the password hashing process to make cracking much harder in case of a breach.

Possible Controls to Make Cracking Harder:

1. **Salting**:
    - **Salting** involves adding random data to each password before it is hashed. This makes sure that even if two users have the same password, their hashes will be different.
    - **Why it helps**: Without a salt, attackers can use precomputed tables (called **rainbow tables**) to crack multiple passwords at once. Salting prevents this by adding randomness, making each password unique.
2. **Using Stronger Hashing Algorithms**:
    - Use algorithms like **bcrypt**, **Argon2**, or **PBKDF2**, which are designed to be slow and resistant to brute-force attacks.
    - **Why it helps**: Slower hashing algorithms take more time to compute, meaning attackers can try fewer guesses per second, making cracking more difficult.
3. **Increase Hashing Iterations**:
    - **Iterations** refer to how many times the hashing algorithm is run on the password. Increasing the number of iterations makes it take longer to compute the hash.

- **Why it helps**: More iterations increase the time it takes to crack each password, slowing down attackers.

4. **Use Multi-Factor Authentication (MFA)**:
   - **MFA** requires users to provide additional information (like a code sent to their phone) besides just the password.
   - **Why it helps**: Even if the password is cracked, MFA adds an extra layer of protection, making it harder for hackers to gain access.
5. **Password Complexity**:
   - Enforcing **strong passwords** (e.g., requiring a mix of uppercase letters, lowercase letters, numbers, and special characters).
   - **Why it helps**: Stronger passwords are harder to guess, even with brute-force or dictionary attacks.
6. **Encryption of the Password Database**:
   - **Encrypting the database** that stores passwords can protect the hashes from being readable even if an attacker gets access to the database.
   - **Why it helps**: If attackers get access to the database but can't decrypt the password hashes, it adds an extra layer of security.
7. **Regular Password Changes and Monitoring**:
   - Encourage users to **change passwords regularly** and monitor for any suspicious activity.
   - **Why it helps**: If a password is compromised, the sooner it's changed, the less chance attackers have of exploiting it.

In Short:

The question is asking what **extra measures** can be put in place to make it **harder for hackers to crack passwords** if the password database is leaked. Measures like salting, using stronger hashing methods, and enabling multi-factor authentication can make it much more difficult for attackers to succeed.

This question is asking you to evaluate the **password policy** of an organization, which refers to the rules and guidelines the organization sets for creating and managing passwords. Specifically, it wants you to focus on aspects like:

Key Aspects of a Password Policy:

1. **Password Length**:
   - **What it means**: This refers to how long a password should be. A longer password is generally more secure because it has more possible combinations.
   - **Why it matters**: Short passwords are easier to guess or crack with brute-force attacks. A good policy should require **at least 8-12 characters** for a password, though more is better (e.g., 16+ characters).
2. **Password Complexity**:
   - **What it means**: This refers to the rules for creating a password that is hard to guess, such as requiring a mix of uppercase letters, lowercase letters, numbers, and special characters.

    - **Why it matters**: Passwords that are more complex are harder to guess or crack. For example, a password like "Password123!" is much harder to guess than "password" or "12345".

3. **Key Spaces**:
    - **What it means**: **Key space** refers to the total number of possible password combinations based on the length and complexity of the password. A larger key space means there are more possible combinations, making it harder for attackers to guess or crack the password.
    - **Why it matters**: For example, a password that is only 6 characters long with only lowercase letters has a small key space (26^6 possible combinations), while a password that is 12 characters long with letters, numbers, and special characters has a much larger key space, making it far harder to crack.
4. **Password Expiry**:
    - **What it means**: Some organizations require users to change their passwords after a set period (e.g., every 90 days).
    - **Why it matters**: Regularly changing passwords reduces the risk of someone using a stolen or cracked password for too long.
5. **Password Reuse Restrictions**:
    - **What it means**: This refers to the rule that prevents users from reusing old passwords.
    - **Why it matters**: Reusing passwords can be risky because if an attacker gets access to an old password, they might be able to use it again. Good policies prevent reuse.
6. **Account Lockouts or Delays**:
    - **What it means**: Some organizations lock accounts or introduce delays after a certain number of failed login attempts to prevent brute-force attacks.
    - **Why it matters**: This helps prevent attackers from trying many passwords in a short time, reducing the risk of successful attacks.

What This Question Wants:

You are being asked to assess or **describe the strength** of the organization's password policy based on these factors. Specifically, the question wants to know:

- **How strong are the passwords required by the policy** (e.g., password length, complexity)?
- **How well does the policy protect against common password attacks** (e.g., brute force, dictionary attacks)?
- **How easy or hard is it for someone to crack or guess the password** based on the policy?

In short, you're evaluating whether the organization's password rules make it **difficult for attackers** to guess or crack passwords. A good password policy should encourage long, complex, and unique passwords to ensure strong security.

- Salts are related to cryptographic nonces?

In very simple words, a **cryptographic nonce** is a **unique number** used only once in a secure communication. The word "nonce" stands for "number used once." It's like a secret code that ensures each transaction or message is unique and can't be reused or tampered with2.

- In practice, a salt is usually generated using a Cryptographically Secure PseudoRandom Number Generator. CSPRNGs are designed to produce unpredictable random numbers which can be alphanumeric. While generally discouraged due to lower security, some systems use timestamps or simple counters as a source of salt. Sometimes, a salt may be generated by combining a random value with additional information, such as a timestamp or user-specific data, to ensure uniqueness across different systems or time periods.

Salt length

If a salt is too short, an attacker may precompute a table of every possible salt appended to every likely password. Using a long salt ensures such a table would be prohibitively large.[7][8] 16 bytes (128 bits) or more is generally sufficient to provide a large enough space of possible values, minimizing the risk of collisions (i.e., two different passwords ending up with the same salt).

Why is Reusing the Same Salt Dangerous?

Using the **same salt** for all passwords is risky for these reasons:

- **Precomputed Tables**: Hackers can create a table of precomputed hashes that include the salt. This means if they get your salt and one hashed password, they can use this table to easily find the original password, making the salt useless.
- **Efficiency of Attacks**: Creating such tables is computationally expensive if each password has a unique salt. But if the same salt is reused, hackers can efficiently build a table that works for all passwords with that common salt.

Problem with Salt Reuse

1. **Same Hash for Same Passwords**: If multiple users have the same password and the same salt is reused, the hashed passwords will look identical.
2. **Single Point of Failure**: If a hacker cracks one password, they can potentially crack others with the same salt and hash pattern.

Password Cracking in computer security is the process of guessing passwords to gain access. **Brute-force attacks** involve repeatedly trying guesses against a password's cryptographic hash. **Password spraying** uses common passwords slowly to avoid detection.

Purposes of password cracking include:

- Recovering a forgotten password
- Gaining unauthorized access
- Checking for weak passwords by system admins
- Accessing restricted digital evidence allowed by a judge

FPGA (Field-Programmable Gate Array)

- **Definition**: An FPGA is a type of hardware that can be programmed or reconfigured by the user after manufacturing.

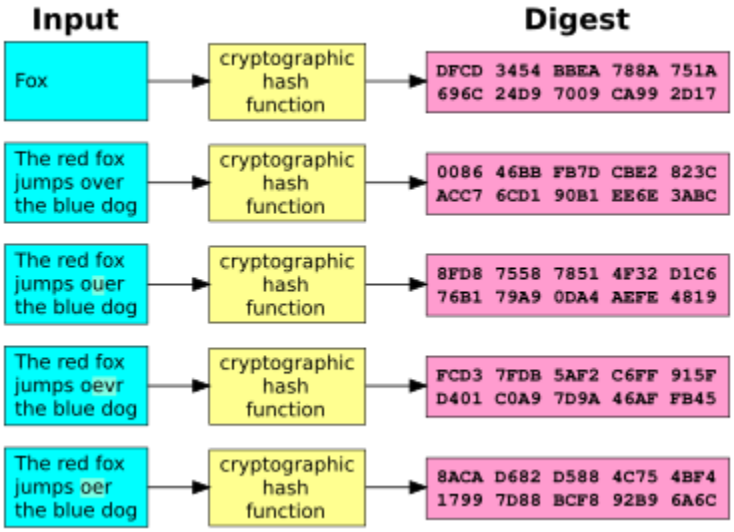
- **Use**: FPGAs are used for various applications that require custom hardware solutions. They are flexible and can be adapted for different tasks, including password cracking.
- **Advantages**:
    - **Flexibility**: Can be reprogrammed for different algorithms or tasks.
    - **Speed**: Faster than general-purpose CPUs for specific tasks due to hardware customization.
    - **Development**: Easier and cheaper to develop for small quantities or specific tasks.
- **Disadvantages**:
    - **Efficiency**: Generally less power-efficient compared to ASICs.
    - **Performance**: While fast, not as fast as ASICs for specific tasks in large-scale deployment.
- **Example Use**: In password cracking, FPGAs can be configured to run specific hashing algorithms much faster than CPUs, making them useful for cracking complex passwords.

ASIC (Application-Specific Integrated Circuit)

- **Definition**: An ASIC is a type of hardware designed for a specific application. Once manufactured, its functionality cannot be changed.
- **Use**: ASICs are used in applications where high efficiency, speed, and large-scale deployment are critical. They are specifically designed for the task they perform.
- **Advantages**:
    - **Efficiency**: Highly power-efficient due to their specific design.
    - **Performance**: Extremely fast for the tasks they are designed for, outperforming both CPUs and FPGAs.
    - **Scalability**: Better suited for large-scale production and deployment.
- **Disadvantages**:
    - **Cost**: High initial development and manufacturing costs.
    - **Flexibility**: Cannot be reprogrammed; designed for a single purpose.
- **Example Use**: In password cracking, ASICs can crack passwords extremely quickly due to their optimized design. An example is the "Deep Crack" machine built by the Electronic Frontier Foundation, which was highly efficient at breaking DES encryption.

Use in Password Cracking

- **FPGA**: Suitable for scenarios where different hashing algorithms need to be tested or where the hardware might need to be repurposed for different tasks.
- **ASIC**: Ideal for large-scale, high-speed password cracking operations where the algorithm is well-defined and unchanging.



A cryptographic hash function (specifically SHA-1) at work. A small change in the input (in the word "over") drastically changes the output (digest). This is called the avalanche effect.

Cryptographic Hash Function (CHF) Summary

Definition: A cryptographic hash function (CHF) is an algorithm that transforms any input (binary string) into a fixed-size output (hash value) with special properties useful for cryptographic applications.

Key Properties:

1. **Fixed-size Output**:
    - The hash value has a fixed size, typically n bits.
    - Probability of a specific n-bit output is 2^{-n}
2. **Preimage Resistance**:
    - Finding an input that matches a specific hash value is infeasible.
    - Security strength is quantified as n bits for an n-bit hash.
3. **Second Preimage Resistance**:
    - Difficulty in finding a different input that produces the same hash value when one input is known.
    - Expected strength is the same as preimage resistance.
4. **Collision Resistance**:
    - Finding any two different inputs that produce the same hash value is infeasible.
    - Strength is typically n/2 bits, due to the birthday paradox.

Applications:

- **Digital Signatures**: Ensures data integrity and authenticity.
- **Message Authentication Codes (MACs)**: Verifies the integrity and authenticity of a message.
- **Authentication**: Verifies the identity of users or devices.
- **Ordinary Hash Functions**:
    - Indexing data in hash tables.
    - Fingerprinting: Identifying duplicate data or files.

- Checksums: Detecting accidental data corruption.

Non-Cryptographic Hash Functions:

- Used for hash tables and error detection.
- Not secure against deliberate attacks, such as denial-of-service (DoS) attacks using easily found collisions (e.g., CRC functions).

Example Uses in Security:

1. **Digital Fingerprints**: Ensuring the integrity of files.
2. **Checksums**: Verifying data integrity.
3. **Preventing Collisions**: Protecting against attacks by ensuring unique hash values for different inputs.

Additional Notes:

- Cryptographic hash functions are distinct from non-cryptographic hash functions, which lack deliberate attack resistance.
- They are essential in various security protocols and applications, ensuring data integrity and security.

FINAL EXAM:

You must determine the following: What type of hashing algorithm was used to protect passwords? What level of protection does the mechanism offer for passwords? What controls could be implemented to make cracking much harder for the hacker in the event of a password database leaking again? What can you tell about the organization's password policy (e.g. password length, key space, etc.)? What would you change in the password policy to make breaking the passwords harder? Enter your memo in the text box below. It should be about 400 words and explain your findings and conclusions of controls currently used by an organization to prevent successful cracking of passwords and potential uplifts that you would propose to existing controls with justifications.

1. What type of hashing algorithm was used to protect passwords?

The hashes in the password dump are using the **MD5** hashing algorithm. This can be identified by the length and format of the hashes (32 hexadecimal characters).

2. What level of protection does the mechanism offer for passwords?

MD5 provides a **low level of protection** for passwords. It is an outdated and insecure hashing algorithm because:

- **Fast Computation**: MD5 is very fast, making it vulnerable to brute-force and dictionary attacks.
- **Collisions**: There are known techniques to find different inputs that produce the same hash (collisions).
- **Precomputed Attacks**: Tools like rainbow tables can quickly find the original input for many MD5 hashes.

3. What controls could be implemented to make cracking much harder for the hacker in the event of a password database leaking again?

To increase security, the following controls should be implemented:

- **Use a Stronger Hashing Algorithm**: Switch to more secure algorithms like bcrypt, scrypt, or Argon2. These are designed to be computationally intensive, slowing down brute-force attacks.
- **Salting**: Use unique salts for each password. This ensures that even if two users have the same password, their hashes will be different.
- **Pepper**: Add a secret value (pepper) to the passwords before hashing them. This value is kept secret and adds an extra layer of security.

4. What can you tell about the organization's password policy (e.g. password length, key space, etc.)?

Based on the cracked passwords, it appears that the organization may not have a strong password policy. The passwords are short, common, and lack complexity:

- **Short Length**: Many passwords are relatively short (e.g., "123456", "password").
- **Common Patterns**: They include common patterns and words that are easily guessable.
- **Lack of Complexity**: Many passwords do not include a mix of uppercase letters, lowercase letters, numbers, and special characters.

5. What would you change in the password policy to make breaking the passwords harder?

To enhance the password policy, the following changes are recommended:

- **Minimum Length**: Require a minimum password length of at least 12 characters.
- **Complexity Requirements**: Enforce the use of uppercase letters, lowercase letters, numbers, and special characters.
- **Prohibit Common Passwords**: Prevent users from using common passwords (e.g., "123456", "password").
- **Regular Updates**: Implement regular password updates and expiration policies.
- **Multi-Factor Authentication (MFA)**: Require MFA for an added layer of security.

WHAT I LEARNED AND DID?

Goldman Sachs Software Engineering Virtual Experience Program on Forage - December 2024 * Completed a job simulation as a Goldman Sachs governance analyst responsible for assessing IT security and suggesting improvements. * Identified that the company was using an outdated password hashing algorithm by cracking passwords using Hashcat. * Wrote a memo for my supervisor summarizing a range of proposed uplifts to increase the company's level of password protection including extending minimum password length and using a dedicated hashing algorithm.

I really enjoyed learning more about [company name]'s culture through the [simulation name] name]. These aspects stood out to me [insert what you liked]. I've done [simulation name] and I

understand that as a [job name] I'll be working on [insert relevant tasks] and this excites me because [insert reason]. I've done [simulation name] and I've now practiced [these skills] which I know are important to succeed as a [job name] at your company.

INTERVIEW TIPS:--

I really enjoyed learning more about Goldman Sachs' culture through the Software Engineering Virtual Experience Program on Forage. The commitment to innovation and continuous learning, as well as the collaborative and inclusive work environment, stood out to me.

I've completed the Software Engineering Virtual Experience Program, and I understand that as a software engineer at Goldman Sachs, I'll be working on developing and maintaining financial systems, ensuring the security of applications, and collaborating with cross-functional teams. This excites me because I am passionate about technology and finance, and I thrive in dynamic and challenging environments where I can continuously learn and grow.

Through the Software Engineering Virtual Experience Program, I have now practiced critical skills such as password cracking, security assessment, and proposing improvements for IT security. I know these skills are important to succeed as a software engineer at your company, and I am eager to apply them in a real-world setting.

“Why are you interested in this role?”

I recently participated in Goldman Sachs’ job simulation on Forage and it was incredibly useful to understand what it might be like to work as a governance analyst there. This simulation enabled me to build my experience cracking passwords using a range of tools including Hashcat. Once I had cracked passwords I was able to come up with a range of uplifts to suggest that would increase the overall level of password protection across the organization. Through this experience, I’ve validated that I really enjoy working on highly impactful technical projects in great detail. I can see just how this project would have a significant impact on Goldman Sachs as a company and on its customers by increasing the level of security to reduce the risk of a successful hack. In future, I would love to apply what I’ve learned during this job simulation in a governance team at Goldman Sachs.

Here are a few key skills you might consider including that are most relevant to the certification and the tasks you completed:

1. **Password Cracking**
2. **Cryptography basics**
3. **IT Security**