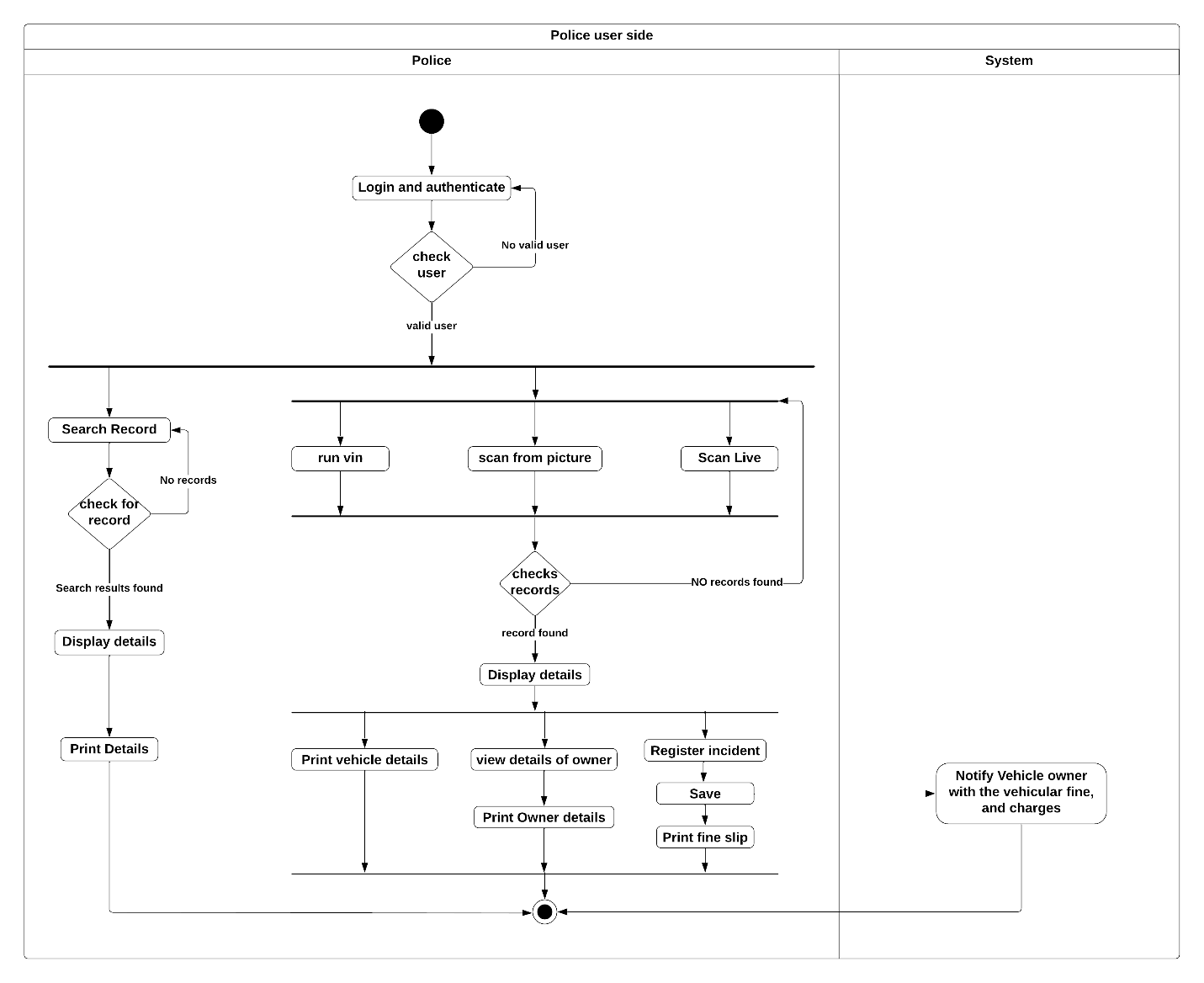
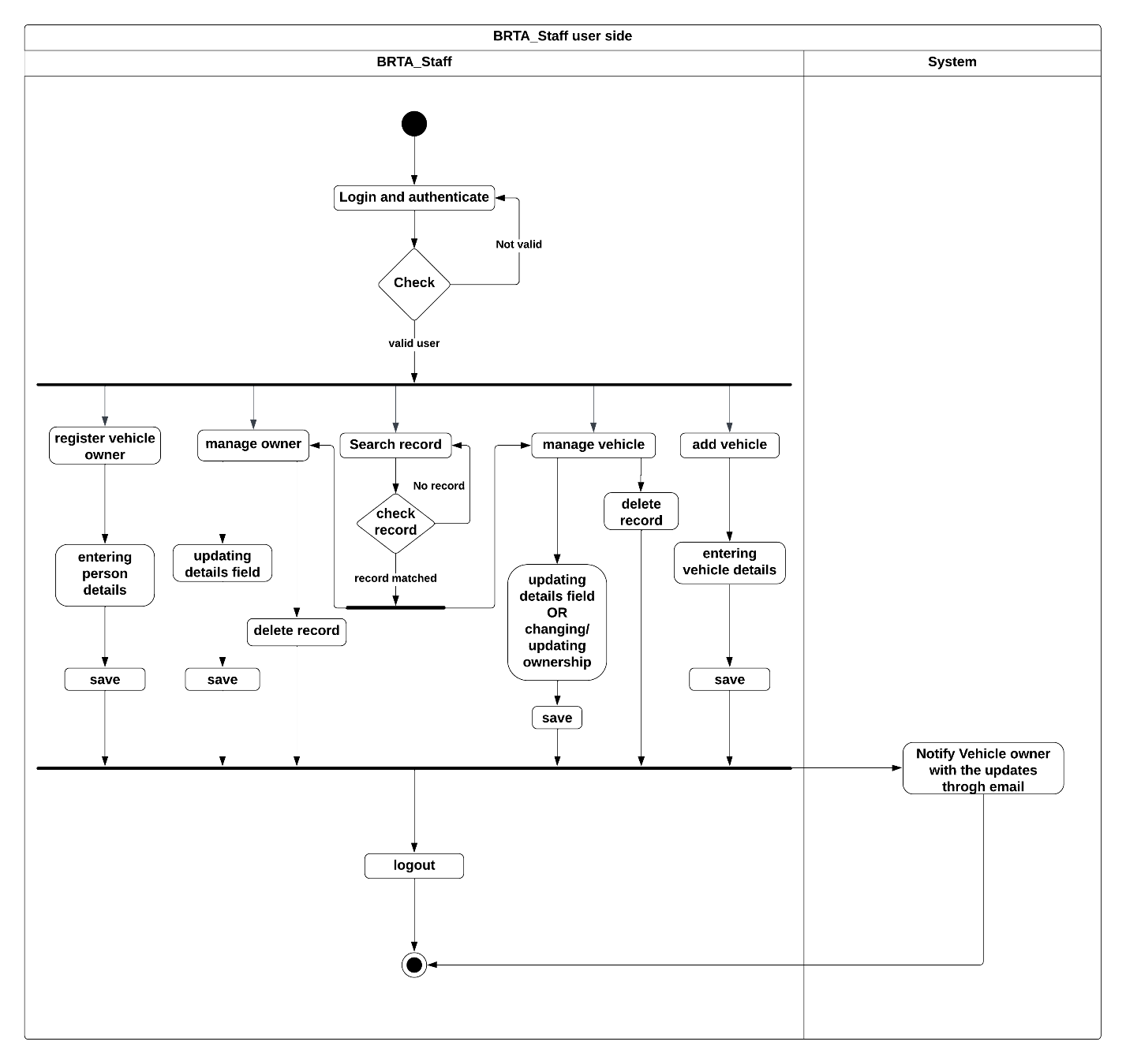
**Activity Diagram of License Plate Recognition and Tracking System**

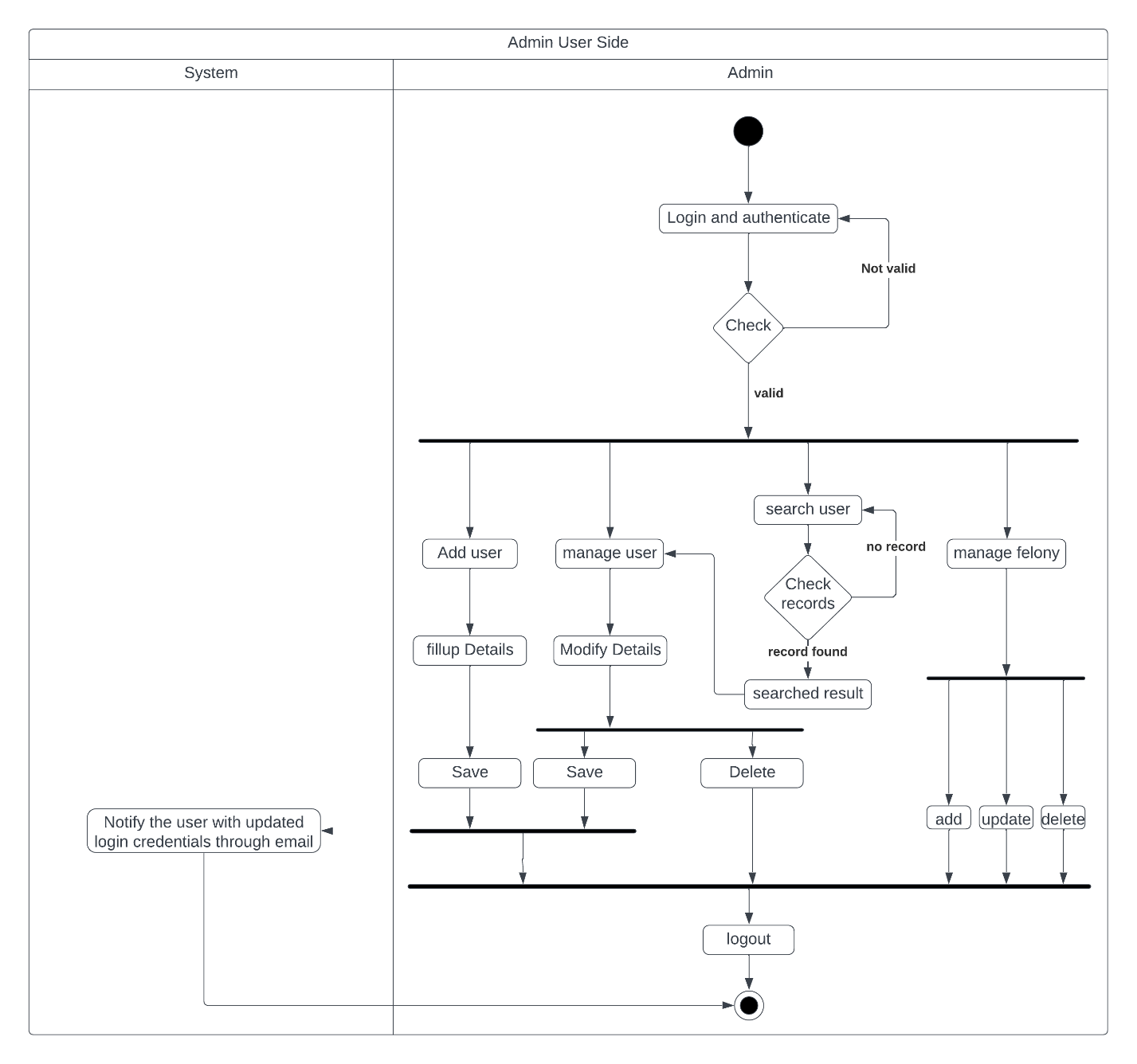
Police User side diagram:



BRTA\_Staff user side diagram:

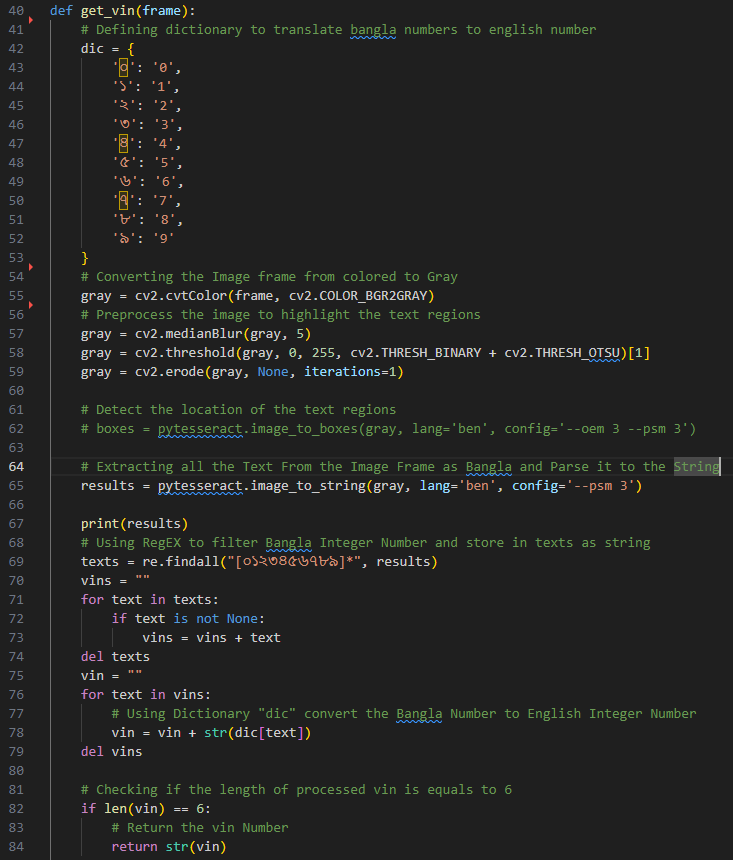


Admin user side diagram:



**Code, Backend Programming:**

**Reading Image and Extracting the Vin number out of the Image frame:**



**Python’s Built in module of SMTP server for sending email.**

A screen shot of a computer program

Description automatically generated with low confidence

**Password storing and login algorithm.**

In the current system we used Django Web Framework which provides [**PASSWORD\_HASHERS**](https://docs.djangoproject.com/en/4.2/ref/settings/#std-setting-PASSWORD_HASHERS) a list of password hashing algorithms.

Django chooses the algorithm to use by consulting the [**PASSWORD\_HASHERS**](https://docs.djangoproject.com/en/4.2/ref/settings/#std-setting-PASSWORD_HASHERS) setting. For storing passwords, Django will use the first hasher in [**PASSWORD\_HASHERS**](https://docs.djangoproject.com/en/4.2/ref/settings/#std-setting-PASSWORD_HASHERS). To store new passwords with a different algorithm, put your preferred algorithm first in [**PASSWORD\_HASHERS**](https://docs.djangoproject.com/en/4.2/ref/settings/#std-setting-PASSWORD_HASHERS)

For verifying passwords, Django will find the hasher in the list that matches the algorithm name in the stored password. If a stored password names an algorithm not found in [**PASSWORD\_HASHERS**](https://docs.djangoproject.com/en/4.2/ref/settings/#std-setting-PASSWORD_HASHERS), trying to verify it will raise **ValueError**.

PASSWORD\_HASHERS **=** [

"django.contrib.auth.hashers.PBKDF2PasswordHasher",

"django.contrib.auth.hashers.PBKDF2SHA1PasswordHasher",

"django.contrib.auth.hashers.Argon2PasswordHasher",

"django.contrib.auth.hashers.BCryptSHA256PasswordHasher",

"django.contrib.auth.hashers.ScryptPasswordHasher",

]

This means that Django will use PBKDF2 to store all passwords but will support checking passwords stored with PBKDF2SHA1, argon2, and bcrypt.

The current process of storing passwords involves several steps, including password hashing, salting, and iteration. Let's take a closer look at the hashing process:

1. **Salt Generation**: A random salt value is generated for each password. The **salt** is a ***cryptographic*** value that adds uniqueness to each password hash, making it more resistant to attacks like rainbow table attacks. Django uses a secure random number generator to generate a **salt**.
2. Iterations: Django performs multiple iterations of the hashing algorithm to slow down the computation process. This is done to make it more computationally expensive and time-consuming for potential attackers trying to guess passwords through brute-force or dictionary attacks.
3. Hashing Algorithm: Django utilizes a secure hashing algorithm to convert the password and salt into a hash. The default algorithm is “**Argon2”** which provides a strong level of security. However, Django also supports other algorithms like **bcrypt, Scrypt** and **PBKDF2** which can be configured in the Django settings.
4. Hash Storage: The resulting hash, along with the salt and the number of iterations used, is stored in the database as the user's password. Django uses a specific format to store this information, typically in the format "algorithm$iterations$salt$hash".

When a user attempts to log in, Django follows a similar process to validate the provided password:

1. Retrieve User's Stored Hash: Django retrieves the stored hash, salt, and iterations associated with the user from the database based on their username or email.
2. Hashing and Comparison: The provided password, along with the retrieved salt and iterations, is hashed using the same algorithm. Django then compares this generated hash with the stored hash. If the two hashes match, the password is considered valid, and the user is authenticated.

Django's password storage process ensures that the actual passwords are never stored directly in the database and provides a strong level of security by employing hashing, salting, and iteration. Django's password verification process ensures that the actual passwords are never stored directly in the database. Instead, it retrieves the stored hash, applies the same hashing algorithm to the provided password, and compares the generated hash with the stored hash. This approach protects user passwords even if the database is compromised.

The specific hashing algorithm used for password verification depends on the configuration in your Django project's settings. The default algorithm starting from Django 3.1 is "Argon2," in this system we are using Django 4.1 but you can choose to use other algorithms like bcrypt or PBKDF2 by adjusting the **PASSWORD\_HASHERS** setting.

This approach makes it significantly more difficult for potential attackers to retrieve the original passwords, even if they gain access to the database.

**Argon2**

The Argon2 algorithm is a state-of-the-art password hashing algorithm that was selected as the winner of the Password Hashing Competition in 2015. It is designed to provide high security against various types of attacks, including brute-force, dictionary, and precomputation attacks.

Argon2 takes inspiration from the previous password hashing algorithms, including **PBKDF2**, **BCrypt**, and **SCrypt**, and aims to improve upon their security and performance characteristics. It offers several notable features that make it a strong choice for password hashing:

1. **Memory-Hardness:** Argon2 is designed to be memory-hard, meaning it requires a significant amount of memory to compute the hash. This property makes it more resistant to parallelization and reduces the efficiency of attacks using specialized hardware, such as GPUs or ASICs.
2. **Resistance to Side-Channel Attacks:** Argon2 is designed to be resistant to side-channel attacks, which aim to extract information from the computational process, such as timing or power consumption. By using data-independent memory access patterns, Argon2 mitigates the risks associated with side-channel attacks.
3. **Configurability:** Argon2 offers flexibility in its configuration, allowing you to tune parameters like time cost, memory cost, and parallelism factor. This configurability allows you to adapt the algorithm to match your specific security and performance requirements.
4. **Protection Against Tradeoff Attacks:** Argon2 is resistant to tradeoff attacks, which involve precomputing a table of hashes to quickly find matches. By employing a built-in memory access pattern that is difficult to optimize for tradeoff attacks, Argon2 makes such attacks impractical.

In this current framework, Argon2 is the password hashing algorithm. It provides a strong level of security for storing user passwords, protecting them even if the database is compromised.

Overall, the Argon2 algorithm offers advanced security features and is recommended for password hashing in modern applications where strong security is a priority.

**Cryptanalysis:**

While there is no public cryptanalysis applicable to Argon2d, there are two published attacks on the Argon2i function. The first attack is applicable only to the old version of Argon2i, while the second has been extended to the latest version (1.3).

The first attack shows that it is possible to compute a single-pass Argon2i function using between a quarter and a fifth of the desired space with no time penalty and compute a multiple-pass Argon2i using only *N*/*e* (≈ *N*/2.72) space with no time penalty. According to the Argon2 authors, this attack vector was fixed in version 1.3.

The second attack shows that Argon2i can be computed by an algorithm which has complexity O(*n*7/4 log(*n*)) for all choices of parameters *σ* (space cost), *τ* (time cost), and thread-count such that *n*=*σ*∗*τ*. The Argon2 authors claim that this attack is not efficient if Argon2i is used with three or more passes.However, ***Joël Alwen*** and ***Jeremiah Blocki*** improved the attack and showed that in order for the attack to fail, Argon2i v1.3 needs more than 10 passes over memory.

**Algorithm:**

**Function** Argon2

**Inputs:**

password (**P**): Bytes (0..232-1) Password (or message) to be hashed

salt (**S**): Bytes (8..232-1) Salt (16 bytes recommended for password hashing)

parallelism (**p**): Number (1..224-1) Degree of parallelism (i.e. number of threads) tagLength (**T**): Number (4..232-1) Desired number of returned bytes

memorySizeKB (**m**): Number (8p..232-1) Amount of memory (in [kibibytes](https://en.wikipedia.org/wiki/Kibibyte)) to use iterations (**t**): Number (1..232-1) Number of iterations to perform

version (**v**): Number (0x13)The current version is 0x13 (19 decimal)

key (**K**): Bytes (0..232-1) Optional key (Errata: PDF says 0..32 bytes, RFC says 0..232 bytes) associatedData (**X**): Bytes (0..232-1) Optional arbitrary extra data

hashType (**y**): Number (0=Argon2d, 1=Argon2i, 2=Argon2id)

**Output:**

tag: Bytes (tagLength)The resulting generated bytes, tagLength bytes long

*Generate initial 64-byte block H0.* All the input parameters are concatenated and input as a source of additional entropy. Errata: RFC says H0 is 64-bits; PDF says H0 is 64-bytes. Errata: RFC says the Hash is H^, the PDF says it's ℋ (but doesn't document what ℋ is). It's actually Blake2b.

Variable length items are prepended with their length as 32-bit little-endian integers.

buffer ← parallelism ∥ tagLength ∥ memorySizeKB ∥ iterations ∥ version ∥ hashType

∥ Length(password) ∥ Password

∥ Length(salt) ∥ salt

∥ Length(key) ∥ key

∥ Length(associatedData) ∥ associatedData

H0 ← Blake2b(buffer, 64) *//default hash size of Blake2b is 64-bytes* Calculate number of 1 KB blocks by rounding down memorySizeKB to the nearest multiple of 4\*parallelism [kibibytes](https://en.wikipedia.org/wiki/Kibibyte)

blockCount ← Floor(memorySizeKB, 4\*parallelism)

Allocate two-dimensional array of 1 KiB blocks (parallelism rows x columnCount columns)

columnCount ← blockCount / parallelism; //In the RFC, columnCount is referred to as **q**

Compute the first and second block (i.e. column zero and one ) of each lane (i.e. row)

**for** i ← 0 **to** parallelism-1 **do** for each row

Bi[0] ← Hash(H0 ∥ 0 ∥ i, 1024) *//Generate a 1024-byte digest*

Bi[1] ← Hash(H0 ∥ 1 ∥ i, 1024) *//Generate a 1024-byte digest*

Compute remaining columns of each lane

**for** i ← 0 **to** parallelism-1 **do** //for each row

**for** j ← 2 **to** columnCount-1 **do** //for each subsequent column

//i' and j' indexes depend if it's Argon2i, Argon2d, or Argon2id (See section 3.4)

i′, j′ ← GetBlockIndexes(i, j) //the GetBlockIndexes function is not defined

Bi[j] = G(Bi[j-1], Bi′[j′]) //the G hash function is not defined

Further passes when iterations > 1

**for** nIteration ← 2 **to** iterations **do**

**for** i ← 0 **to** parallelism-1 **do** for each row

**for** j ← 0 **to** columnCount-1 **do** //for each subsequent column

//i' and j' indexes depend if it's Argon2i, Argon2d, or Argon2id (See section 3.4) i′, j′ ← GetBlockIndexes(i, j)

**if** j == 0 **then**

Bi[0] = Bi[0] xor G(Bi[columnCount-1], Bi′[j′])

**else**

Bi[j] = Bi[j] xor G(Bi[j-1], Bi′[j′])

Compute final block **C** as the XOR of the last column of each row

C ← B0[columnCount-1]

**for** i ← 1 **to** parallelism-1 **do**

C ← C **xor** Bi[columnCount-1]

Compute output tag

**return** Hash(C, tagLength)

**Variable-length hash function:**

Argon2 makes use of a hash function capable of producing digests up to 232 bytes long. This hash function is internally built upon [Blake2](https://en.wikipedia.org/wiki/Blake2).

**Function** Hash(message, digestSize)

**Inputs:**

message: Bytes (0..232-1) Message to be hashed

digestSize: Integer (1..232) Desired number of bytes to be returned

Output:

digest: Bytes (digestSize)The resulting generated bytes, digestSize bytes long

**Hash** is a variable-length hash function, built using Blake2b, capable of generating digests up to 232 bytes.

If the requested digestSize is 64-bytes or lower, then we use Blake2b directly

**if** (digestSize <= 64) **then**

**return** Blake2b(digestSize ∥ message, digestSize) //concatenate 32-bit little endian digestSize with the message bytes

For desired hashes over 64-bytes (e.g. 1024 bytes for Argon2 blocks), we use Blake2b to generate twice the number of needed 64-byte blocks, and then only use 32-bytes from each block

Calculate the number of whole blocks (knowing we're only going to use 32-bytes from each)

r ← Ceil(digestSize/32)-2;

Generate r whole blocks.

Initial block is generated from message

V1 ← Blake2b(digestSize ∥ message, 64);

Subsequent blocks are generated from previous blocks

**for** i ← 2 **to** r **do**

Vi ← Blake2b(Vi-1, 64)

Generate the final (possibly partial) block

partialBytesNeeded ← digestSize – 32\*r;

Vr+1 ← Blake2b(Vr, partialBytesNeeded)

Concatenate the first 32-bytes of each block Vi

(except the possibly partial last block, which we take the whole thing)

Let Ai represent the lower 32-bytes of block Vi

**return** A1 ∥ A2 ∥ ... ∥ Ar ∥ Vr+1

**PBKDF2**

**Technically, the input data for PBKDF2 consists of:**

* password – array of bytes / string, e.g. "*p@$Sw0rD~3*" (8-10 chars minimal length is recommended)
* salt – securely-generated random bytes, e.g. "*df1f2d3f4d77ac66e9c5a6c3d8f921b6*" (minimum 64 bits, 128 bits is recommended)
* iterations-count, e.g. 1024 iterations
* hash-function for calculating **HMAC**, e.g. SHA256
* derived-key-len for the output, e.g. 32 bytes (256 bits)

The **output data** is the **derived key** of requested length (e.g. 256 bits).

**PBKDF2 and Number of Iterations**

**PBKDF2** allows to configure the number of **iterations** and thus to configure the time required to derive the key.

* **Slower key derivation** means high login time / slower decryption / etc. and **higher resistance** to password cracking attacks.
* **Faster key derivation** means short login time / faster decryption / etc. and **lower resistance** to password cracking attacks.
* **PBKDF2** is **not resistant** to [GPU attacks](https://security.stackexchange.com/questions/118147/how-are-gpus-used-in-brute-force-attacks) (parallel password cracking using video cards) and to [ASIC attacks](https://en.wikipedia.org/wiki/Custom_hardware_attack) (specialized password cracking hardware). This is the main motivation behind more modern KDF functions.

**CSRF (Cross Site Request Forgery)**

Tokens can be a great mechanism in preventing CSRF attacks.

A CSRF Token is a secret, unique and unpredictable value a server-side application generates in order to protect CSRF vulnerable resources. k

The tokens are generated and submitted by the server-side application in a subsequent HTTP request made by the client. After the request is made, the server side application compares the two tokens found in the user session and in the request. If the token is missing or does not match the value within the user session, the request is rejected, the user session terminated, and the event logged as a potential CSRF attack.

**How should CSRF tokens be generated?**

Just like session tokens in general, CSRF tokens should contain significant entropy and be strongly unpredictable.

You can achieve this by using a cryptographic strength pseudo-random number generator (PRNG), seeded with the timestamp when it was created and a static secret.

For further security, you can generate individual tokens by chaining their outputs with user-specific entropy and take a strong hash of the whole structure.

This presents an additional obstacle to a malicious user who attempts to analyze the tokens based on a sample that is issued to him.

In short, here are the principles you should follow when generating and verifying your token:

* Use a well-established random number generator with enough entropy
* Make sure tokens can’t be reused. Expire them after a short amount of time
* Verify the received token is the same as the set token in a safe way, for example, compare hashes
* Do not send CSRF tokens in HTTP GET requests. This will make sure they are not directly available in the URL and they don’t leak in the Referer header with other referrer information

For example, a CSRF token in PHP can be generated as follows:

**$\_SESSION[‘token’] = bin2hex(random\_bytes(24));**

And verify the token as follows:

if(hash\_equals($\_SESSION[‘token’], $\_POST[‘token’])) {  
 // Action if the token is valid  
}

else {  
 // Action if the token is invalid  
}

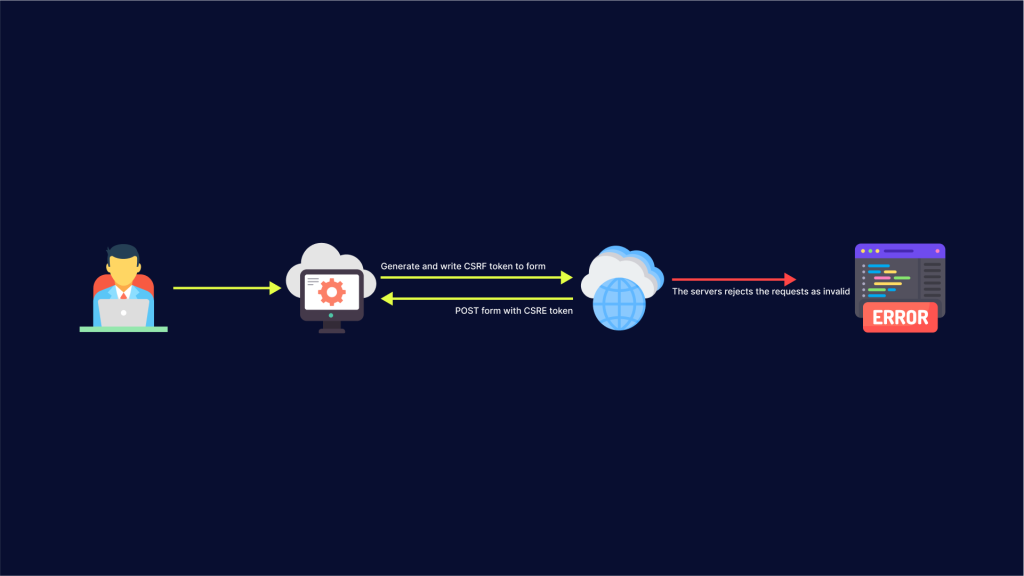
If you prefer a more secure approach, generate separate tokens for each form. Make sure you don’t expose the token directly to the user’s browser. Hash the token using the filename of the form Hash the token with the filename of the form. Here is one example how to do it in PHP:  
  
hash\_hmac(‘sha256’, ‘post.php’, $\_SESSION[‘internal\_token’])

When you verify, compare the hashes. If both the token and the form are valid, the hashes will match.

**How should CSRF tokens be transmitted?**

CSRF tokens are secrets and should be handled as such in a secure manner throughout their lifecycle.

Try transmitting the token to the client within a hidden HTML form field, using the POST method. This way the token will be included as a request parameter when the form is submitted.



For example:

<form action="" method="post"> {% csrf\_token %}

<input type="hidden" name="CSRFToken" value="OWY4NmQwODE4ODRjN2Q2NTlhMmZlYWEwYzU1YWQwMTVhM2JmNGYxYjJiMGI4MjJjZDE1ZDZMGYwMGEwOA==">

[...]

</form>

**CSRF protection in Django**

Django is a free backend framework based on Python. Django focuses on reusability of code and pluggability of modules along with low coupling and rapid development principles.

Django offers middleware for protecting a web server against CSRF attacks. To protect your apps, the middleware must be activated in your project. Also, you have to include the csrf\_token tag inside the form elements which point to any in-project URLs.