SCRAM Report

Subject	Data Integrity and Authentication
Type	Project

📝 Detailed Breakdown of the SCRAM Authentication Showcase

This showcase demonstrates how the **SCRAM (Salted Challenge Response Authentication Mechanism)** protocol works to authenticate a user securely — without ever transmitting or storing the user's password directly. The following detailed explanation walks through the login flow shown in your UI.

👱 Inputs from the UI

Field	Value
Username	mohamed
Password	mohamed123
Salt (Base64)	cLdWz8jgKEbVbkFa9RBTQQ==
Iterations	4096
Client Nonce	VT6AmDL8Nfx7dSiw
Server Nonce	hnnZuR/2K0w6SObPJsNwBw==

These values are used to derive secure keys and prove the user's identity.

> While the login process involves verifying a user's knowledge of their password-derived secrets, the registration process is where those secrets are first created and securely stored. During registration, the client provides a username and password, and the server generates a unique salt and iteration count. It then derives a salted password using PBKDF2, computes the client key, stored key, and server key, and stores only the derived values (not the raw password). In contrast, during login, these same values are re-derived on the client-side and compared indirectly by the server using a proof, ensuring secure authentication without ever transmitting the actual password.

🔐 Step-by-Step: What Happens During Login

Step 1: Send `/auth/start` Request

```
POST /auth/start
{
  username: "mohamed",
  clientNonce: "VT6AmDL8Nfx7dSiw"
}
```

- **What it does:** Begins the login process by sending the username and a client-generated nonce.
- **Server responds with**:

  ```
  {
    salt: "...",
    iterations: 4096,
    serverNonce: "...",
    combinedNonce: clientNonce + serverNonce
  }
  ```

- **Why it's important:**
  - The server provides the salt and iteration count used to derive the keys.
  - The nonce combination prevents replay attacks.

Step 2: Derive the Salted Password

```
saltedPassword = PBKDF2(password, salt, iterations)
```

- **What it does:** Hashes the user's raw password with a server-provided **salt** using **PBKDF2** (a slow, secure key derivation function), repeated 4096 times.
- **Why it's important:**
  - Makes brute-force and rainbow table attacks impractical.
  - Salt ensures two identical passwords produce different results.
- **Result**:

  ```
  hbvGYZGISkjXITVgsmlUcZVn+t6DeB/gJig7USQbeGg=
  ```

Step 3: Generate the Client Key

```
clientKey = HMAC(saltedPassword, "Client Key")
```

- **What it does:** Uses the salted password as a key in an HMAC with the string `"Client Key"` as the message.
- **Why it's important:** Creates a unique "client identity key" tied to the user's password.
- **Result**:

  ```
  POJsPrKARg507ZEJUm0q/Wci0iEZLbH9CJsrT7ddID4=
  ```

Step 4: Derive the Stored Key

```
storedKey = SHA256(clientKey)
```

- **What it does:** Hashes the client key using SHA-256.
- **Why it's important:**
  - This is the only client-authenticating value stored in the server database.
  - Prevents the server from needing to keep the raw client key.
- **Result**:

  ```
  9M4v9GNAVjTq27×9kmpsAmCLEF7TD3XEJ2479I8FUqk=
  ```

Step 5: Build the Authentication Message

```
authMessage = "n=username,r=clientNonce,s=salt,i=iterations,r=combinedNonce"
```

- **What it does:** Combines critical information from the client and server into a canonical message.
- **Why it's important:**
  - Both the client and server will use this exact string to compute HMACs.
  - It proves both sides are working on the same values and prevents MITM attacks.
- **Result:**

  ```
  n=mohamed,r=VT6AmDL8Nfx7dSiw,s=cLdWz8jgKEbVbkFa9RBTQQ==,i=4096,r=VT6AmDL8Nfx7dSiwhnnZuR/2K0w6SObPJsNwBw==
  ```

Step 6: Compute the Client Signature

```
clientSignature = HMAC(storedKey, authMessage)
```

- **What it does:** Uses the stored key to HMAC the `authMessage`.
- **Why it's important:**
  - This signature will help the server **reconstruct** the client key (indirectly) and verify that the user knows their password.
- **Result**:

  ```
  ErYuUBUp64NZMQg82ALnHtdzS722rbgNYgAYPv80tcY=
  ```

Step 7: Create the Client Proof

```
clientProof = XOR(clientKey, clientSignature)
```

- **What it does:** XORs the client key with the client signature.
- **Why it's important:**
  - This is what the client **sends** to the server.
  - It safely hides the real `clientKey`, yet allows the server to verify the signature without ever revealing the password.
- **Result (Base64)**:

  ```
  LIRCbqeprY0t3Vu1im/N47v/cZyvQgnw8JKkTLhpPFg=
  ```

Step 8: Derive the Server Key

```
serverKey = HMAC(saltedPassword, "Server Key")
```

- **What it does:** Similar to client key, but using `"Server Key"` as the message.
- **Why it's important:**
  - Only the server knows this value.
  - It will be used to create a server signature and prove the server's authenticity to the client.
- **Result**:

  ```
  +97Fv9aHf2WPcMoFNdC06SEesk+s3sTant2JlmvFtIdw=
  ```

Step 9: Generate the Server Signature

```
serverSignature = HMAC(serverKey, authMessage)
```

- **What it does:** HMAC using the server key and the same `authMessage`.
- **Why it's important:**
  - This will be sent back to the client.
  - The client will compute the same value and compare to ensure it's talking to the real server (prevents spoofing).
- **Result**:

  ```
  c4ZPaq/O8pW2iuLa2821lNf/ckLnx4RM7BlJgul8rylU=
  ```

Step 10: Send `/auth/finish` with Proof

```
POST /auth/finish
{
  username: "mohamed",
  clientProof: "LlRCbqeprY0t3Vu1im/N47v/cZyvQgnw8JKkTLhpPFg="
}
```

- **What it does:** Sends the derived `clientProof` to the server.
- **Why it matters:**
  - The server uses this to verify that the user knew the correct password (without ever seeing it).
  - It reconstructs the client key and verifies against stored credentials.

✅ Final Response

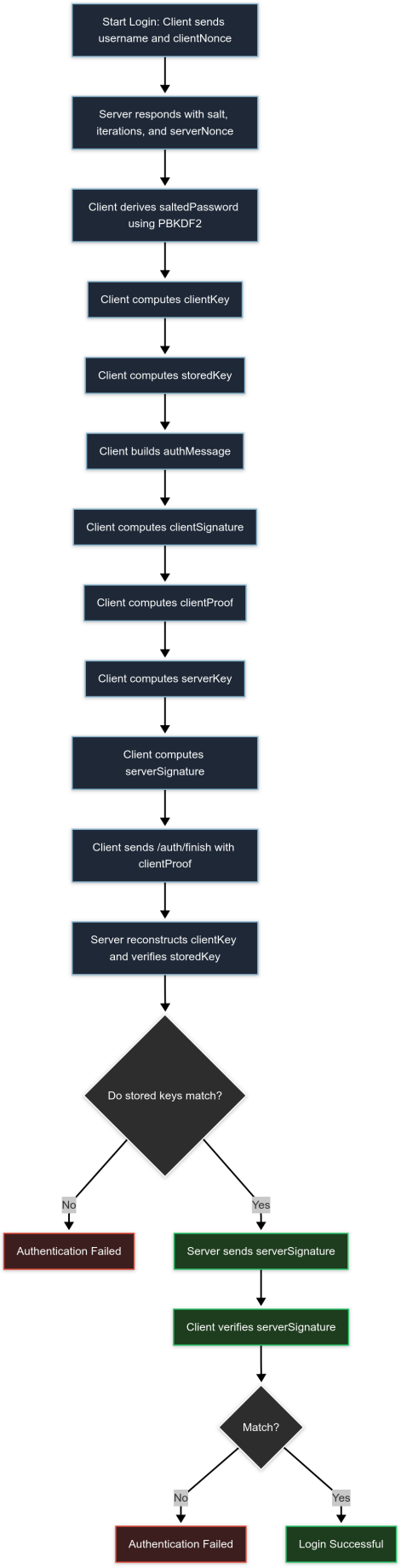
```
{
  message: "Authenticated",
  serverSignature: "c4ZPaq/O8pW2iuLa2821lNf/ckLnx4RM7BlJgul8rylU="
}
```

- **What it means**:
  - The server verified the `clientProof`.
  - It returns a `serverSignature` that proves **it too knows the password-derived secret**.
- **The client should verify this value** by computing its own and matching it.

✅ Summary

Step	Action	Location	Purpose
1	`POST /auth/start` with username & clientNonce	Client → Server	Initiates login; asks server for salt, iteration count, and nonce
2	Receive salt, iterations, and serverNonce	Server → Client	Supplies data needed for key derivation and nonce uniqueness
3	Derive `saltedPassword = PBKDF2(password, ...)`	Client	Transforms the password into a secure key using salt & iterations
4	Compute `clientKey = HMAC(saltedPassword, ...)`	Client	Key used to prove client identity securely
5	Compute `storedKey = SHA256(clientKey)`	Client	Client-side version of what was stored on the server during registration
6	Build `authMessage`	Client	Canonical string combining username, nonces, salt, and iterations
7	Compute `clientSignature = HMAC(storedKey, ...)`	Client	Signature based on storedKey and authMessage
8	Compute `clientProof = XOR(clientKey, signature)`	Client	What the client sends to prove password knowledge
9	Compute `serverKey = HMAC(saltedPassword, ...)`	Client	Prepares for server verification (mutual auth)
10	Compute `serverSignature = HMAC(serverKey, ...)`	Client	Used to verify that the server is genuine
11	`POST /auth/finish` with `clientProof`	Client → Server	Submits proof of password knowledge to the server
12	Server verifies proof & responds with signature	Server	Confirms authentication success and proves server identity

🔃 Flowchart for Login Flow


Start Login: Client sends username and clientNonce
Server responds with salt, iterations, and serverNonce
Client derives saltedPassword using PBKDF2
Client computes clientKey
Client computes storedKey
Client builds authMessage
Client computes clientSignature
Client computes clientProof
Client computes serverKey
Client computes serverSignature
Client sends /auth/finish with clientProof
Server reconstructs clientKey and verifies storedKey
Do stored keys match?
No
Yes
Authentication Failed
Server sends serverSignature
Client verifies serverSignature
Match?
No
Yes
Authentication Failed
Login Successful