APPS BY VIKAS

Meet Vikas



## GenzLudo

Ludo with 3D characters and up to 8 players. A first-of-its-kind no-teaming-up feature keeps play fair. Try it - you'll love the difference.

**Download on Google Play**



## Kachu

Too much scrolling straining your neck? Add your addictive apps and complete daily neck exercises to unlock them.

**Download on Google Play**

WEBSECURITY·PRACTICALGUIDE

# You Added 2FA — But Most Tutorials Miss These 2 Checks

A complete, production-grade guide to protecting your admin panel with TOTP two-factor authentication — whether you log in with **Continue with Google** **email and password**

**By Vikas Yadav**

Written for developers shipping real products, not demos.

Framework-neutral: Next.js, Firebase, Supabase, Django, Laravel, Express.

S TA RT H E R E

# The two checks almost every 2FA tutorial forgets

Most guides teach you how to make the 6-digit code work. Very few teach you how to make it actually safe. These two are the gaps.

If you follow a normal YouTube tutorial, you will end up with 2FA that *looks* correct. The QR code scans. The code verifies. The dashboard opens. And two real vulnerabilities are sitting quietly inside it.

## 01 Replay protection: one code, one use

A TOTP code is valid for a whole 30-second window. If your server only asks “is this code mathematically correct right now?”, then the same code can be submitted more than once inside that window and it will pass every time.

Why that matters in real life:

- Someone glances at your phone screen while you type the code.
- A code gets typed into a phishing page and instantly reused on the real site.
- A code leaks into a log file, a screen recording, or a support chat.

The fix is small. After a code is accepted, remember which time step it belonged to, and refuse anything from that step or earlier.

```
Code accepted
      |
      v
Save last_used_step = 58412903  (per user)
      |
      v
Next attempt arrives
      |
      v
Is incoming step <= last_used_step ?
      |
   YES  -->  REJECT (even if the maths is correct)
      |
    NO   -->  Continue verification
```

**THISISNOTOPTIONAL**

RFC 6238 is explicit about it: once an OTP has been successfully validated, the same OTP must not be accepted a second time. If you implement only one thing from this guide, implement this.

## 02 Session rotation: upgrade the session, don’t flip a flag

Here is the pattern almost everyone writes:

```
Password verified  -->  session created
2FA verified       -->  session.twoFactorVerified = true
```

The problem is that the session which now has admin power is the *same* session that existed before 2FA was completed. Its ID was handed out at a lower privilege level. If that identifier was captured at any point — a fixation attack, a leaked URL, a shared device, a logging mistake — then the attacker’s copy of the session silently gets upgraded to admin the moment you pass 2FA. You did the work; they inherited the result.

The correct shape is: finish the challenge, throw the challenge state away, and issue a fresh session.

```
Google / Email + Password
          |
          v
  Primary auth verified
          |
          v
  Short-lived MFA challenge     <-- NO admin access yet
          |
          v
  TOTP or recovery code
          |
          v
  Backend verifies
          |
          v
  Invalidate the MFA challenge
          |
          v
  Create / rotate the final session   <-- NEW session ID
          |
          v
  Session carries MFA assurance
          |
          v
  Admin APIs allowed
```

> **THE RULE UNDERNEATH**
>
> Whenever the privilege level of a session changes, the session identifier should change with it. This is standard OWASP session-management guidance, and login-plus-MFA is exactly such a change.

Everything from here on is the full build, in order. These two checks are folded into it in their proper places.

BACKGROUND

# How TOTP actually works

Two minutes of theory that makes every later decision obvious.

TOTP stands for Time-based One-Time Password. It is the standard behind Google Authenticator, Microsoft Authenticator, Authy, 1Password and the rest. The mechanism is simpler than people expect.

Your server and the user's app share one secret. Both of them also know the current time. Each side chops time into fixed windows — normally 30 seconds — and numbers them. Then each side independently runs the same calculation:

```
secret + current_time_step
          |
          v
    HMAC-SHA1
          |
          v
    truncate to 6 digits
          |
          v
      482193
```

Both sides get the same six digits without ever talking to each other. That is the entire trick.

> **A COMMON MISCONCEPTION**
>
> You do **not** need a Google API, a Google SDK, or any Google permission just because your user picked Google Authenticator. The app is a generic TOTP calculator. It never contacts your server, and your server never contacts it. They only share a secret.

Three consequences follow directly, and they shape the rest of this guide:

| BECAUSE… | …THIS FOLLOWS |
|---|---|
| The server needs the original secret to compute future codes | You must **encrypt** |
| Clocks drift apart slightly | You must accept a small time window, not one exact step |
| A code stays valid for a whole window | You must block reuse within that window |

**DO NOT WRITE THE ALGORITHM YOURSELF**

Use a maintained TOTP library for your language. Hand-rolled HMAC truncation is a classic source of subtle, silent bugs — and a broken 2FA implementation is worse than none, because you will trust it.

PART 1 · ENROLLMENT

# Turning 2FA on

Setting up the secret, showing the QR code, and proving it works before you commit.

## 03 Add the “Enable 2FA” option

Inside your admin security settings, add a simple state display:

```
Two-Factor Authentication

Status: Disabled

[ Enable 2FA ]
```

Only an already-authenticated admin may start this flow. Better still, ask them to re-enter their password first. Enrolling a second factor is a security-sensitive change, and a stolen browser session should not be enough to perform it.

## 04 Generate the secret on the backend

When the admin clicks Enable, the *server* generates a cryptographically random secret. Not the browser. Not a predictable value derived from the user ID or email.

```
User ID: 123
2FA Enabled: false
Pending TOTP Secret: JBSWY3DPEHPK3PXP...
```

Every user gets their own unique, independently random secret. Never share one secret across accounts, and never derive it from something guessable.

## 05 Store it correctly: encrypt, never hash

This is the single most misunderstood part of building 2FA, because it breaks the habit everyone learned from passwords.

| VALUE | TREATMENT | WHY |
|---|---|---|
| Password | **Hash** | You only ever need to check a match |
| Recovery code | **Hash** | You only ever need to check a match |
| TOTP secret | **Encrypt** | You need the original value back to compute the next code |

Now the part that actually decides whether the encryption was worth anything — where the key lives.

**POINTLESS**

Key stored in the database
Key in the same table as the secret
Key hardcoded in source code
Key committed to the repo

One DB dump
↓
Every admin secret

**CORRECT**

Environment secret
Secret Manager
Cloud KMS / HSM

DB dump alone
↓
Encrypted blobs
(useless)

If the attacker gets your database and your key together, encryption bought you nothing. Separation is the whole point.

## 06 Build the QR code — locally

Your backend turns the secret into a standard `otpauth://` URI:

```
otpauth://totp/MyWebsite:admin@example.com
  ?secret=YOUR_SECRET
  &issuer=MyWebsite
```

**NEVERUSEANONLINEQRGENERATOR**

Look closely at that URI: it contains the actual TOTP secret in plain text. The moment you send it to a third-party QR image API, you have handed a stranger's server the credential that protects your admin panel — and it is now in their logs. Generate the QR with a library inside your own frontend or backend. Always.

Display it together with the secret in text form, so people using a password manager or a desktop authenticator can enrol without a camera:

```
Enable Two-Factor Authentication

Scan this QR code with your
authenticator app.

     [ QR CODE ]

Can't scan? Enter this key manually:
JBSW Y3DP EHPK 3PXP

Enter the 6-digit code:
[ _ _ _ _ _ _ ]

[ Verify & Enable ]
```

## 07 Verify the first code before you enable anything

Do not set `2fa_enabled = true` when you show the QR code. If the scan silently failed, or the phone's clock is badly wrong, you have just locked yourself out of your own admin panel with a working-looking screen.

Keep the secret in a *pending* state and make the admin prove it:

```
POST /api/2fa/verify-setup
code = 482193

          |
          v
Pending secret + current time + entered code
          |
          v
     Verify
          |
  OK --> promote pending secret to active
         2fa_enabled = true
          |
 FAIL --> "Invalid authentication code"
         secret stays pending
```

## 08 Generate recovery codes

Ask the honest question: what happens when you lose your phone? Without an answer, you are permanently locked out of your own product.

```
SAVE YOUR RECOVERY CODES

K7DP-X29Q
M8LK-72WA
P4RT-91ZK
...

Each code works once.
```

Every recovery code must be random, single-use, and **hashed**
here because, unlike the TOTP secret, you only ever need to check whether a submitted code matches.




**THE ENDPOINT EVERYONE FORGETS TO RATE-LIMIT**

Developers carefully rate-limit the TOTP endpoint and then leave the recovery-code endpoint wide open. It is frequently the weaker path into the same account. Rate-limit both, with the same seriousness.

PART 2 · LOGIN

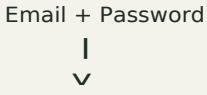
# The login flow, both ways

Email and password, Continue with Google, and the challenge state that sits between them.

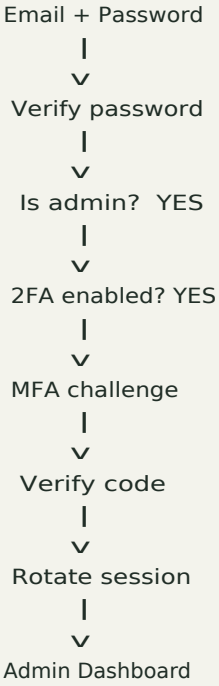
## 09 Email and password

The flow you have now, and the flow you need:

**BEFORE**




**AFTER**




### THE MFA CHALLENGE STATE

Between “password correct” and “fully logged in” there is an intermediate state. Most guides mention it and then say nothing about its properties, which is where implementations go wrong. Give it all four:

| PROPERTY | MEANING |
|---|---|
| **Short-lived** | Expires in about 5 minutes. It is not a session. |
| **Single-use** | Destroyed the moment the challenge succeeds or is abandoned. |
| **Bound** | Tied to that one login attempt and that one user. |
| **Powerless** | Rejected by every normal and admin API. It can call exactly one endpoint: verify. |

Conceptually, while the challenge is open:

```
primary_auth = verified
mfa          = pending

/admin/users      ->  403
/admin/settings   ->  403
/admin/database   ->  403
/api/2fa/verify   ->  allowed
```

Miss the fourth property and you have accidentally created a half-privileged session that skips 2FA entirely. That is the failure mode this state exists to prevent.

## 10 Continue with Google

Google login handles *who the user is*. It does not decide who gets your admin panel — that is your job, and your TOTP challenge sits on top of it.

```
Continue with Google
          |
          v
Google authentication succeeds
          |
          v
Backend validates the Google ID token
          |
          v
Is this an authorised admin?  YES
          |
          v
MFA challenge  -->  TOTP code
          |
          v
Server verifies
          |
          v
Rotate session
          |
          v
Admin Dashboard
```

#### VALIDATING THE TOKEN PROPERLY

“Backend validates Google identity” is where a lot of implementations quietly become insecure. Verify all of it, server-side:

| CHECK | WHY IT MATTERS |
|---|---|
| **Signature** | Proves Google issued the token and it was not forged or edited |
| **aud** | Proves the token was minted for *your* client ID, not some other app's |
| **iss** | Proves the issuer really is Google |
| **exp** | Proves the token has not expired |

**KEY THE ACCOUNT ON SUB, NOT EMAIL**

Use Google's `sub` claim as the stable unique identifier for the account. Email addresses can change, and treating them as the primary identity key creates account-linking problems that turn into security problems.

**IF YOUR PLATFORM ALREADY SUPPORTS MFA**

Firebase Authentication with Identity Platform, Supabase, Auth0 and others ship supported TOTP MFA flows. Use the official flow rather than rebuilding this yourself — the checklist at the end still applies, you are just not writing the verification layer by hand.

## 11 Verifying the code

Three things have to happen here, and skipping any one of them is a real bug.

### A. ACCEPT A SMALL DRIFT WINDOW

Phone clocks and server clocks are never perfectly aligned. If you accept only the exact current step, users will hit "invalid code" while staring at a correct code, and you will get support messages you cannot reproduce.

A small window such as one step either side of the current one (roughly 90 seconds total for a 30-second TOTP) is the common practical configuration. Keep it limited — a wider window means more codes are acceptable at any given moment, which increases exposure and makes strict rate limiting even more important. Also make sure the server itself is NTP-synchronised; a drifting server produces failures for everybody at once.

### B. REJECT REPLAYS

As covered at the start: record the time step of every accepted code, and reject any attempt at that step or earlier. Do this per user.

#### C. COMPARE IN CONSTANT TIME

Use your language's constant-time comparison function for both TOTP codes and recovery codes, rather than `==`. Most good TOTP libraries do this internally; recovery codes are usually your own code, so check that path yourself.

## 12 Rotate the session

The challenge passed. Now, before you grant anything:

```
Verification succeeds
        |
        v
Invalidate the MFA challenge state
        |
        v
Issue a NEW session identifier
        |
        v
Record MFA assurance on the server side
  session.mfa = verified
  session.mfa_at = timestamp
        |
        v
Admin APIs now permitted
```

Two details worth stating plainly. First, a frontend variable saying `twoFactorVerified = true` is decoration, not security — anyone can set that. The server must know that *this specific session*
possible later (section 17).

PART 3 · HARDENING

# Everything around the login

The parts that decide whether your 2FA is real or decorative.

## 13 Protect the backend, not just the page

This is the most important section in the guide, and the one most often skipped.

Hiding `/admin` behind a 2FA screen protects the *page* page talks to:

```
GET    /api/admin/users
DELETE /api/admin/user/123
POST   /api/admin/settings
POST   /api/admin/products
```

If those endpoints only check “is there a session?”, then anyone with a pre-MFA session can call them directly with a simple HTTP client and never see your login screen at all. The UI was never the security boundary.

Every sensitive endpoint independently checks all three:

```
Request
   |
   v
Valid session?       NO --> 401
   |
   v
Admin role?          NO --> 403
   |
   v
MFA completed on
this session?        NO --> 403
   |
   v
Allow request
```

## 14 Rate limit properly

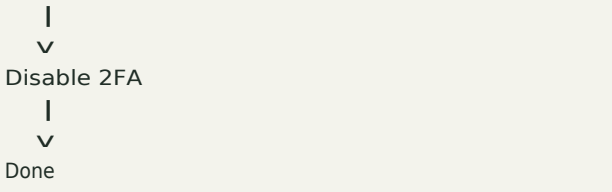
A 6-digit code has a million combinations, and with a drift window several codes are acceptable at any instant. Unlimited guessing turns 2FA into a formality.

```
Wrong code
Wrong code
Wrong code
... unlimited attempts     <-- 2FA is now cosmetic
```

Apply strict attempt limits to *every* verification endpoint: TOTP verification, recovery-code verification, and the enrollment verification step. Lock or back off after a small number of failures, and log repeated failures as a signal — a burst of wrong codes on an admin account is worth an alert.

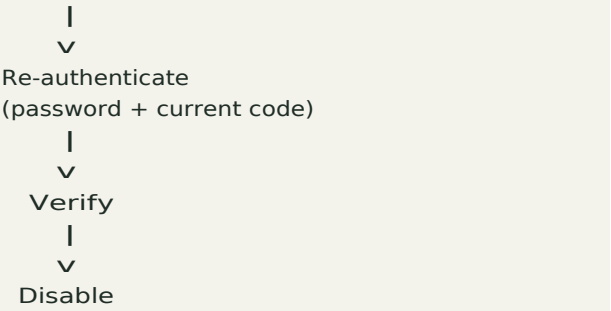
## 15 Protect “Disable 2FA”

**WRONG**




An attacker with a stolen session just switches your protection off.

**RIGHT**




The same applies to replacing or re-enrolling a factor, and to generating fresh recovery codes. An existing session alone is never sufficient authority to change how the account is protected.

## 16 Never log secrets or codes

These lines must not exist anywhere in your system:

```
User entered OTP: 482193
TOTP Secret: JBSWY3DPE...
```

Remember that logs are not just your own log statements. Error reporting tools, request-body capture, analytics, APM traces and crash reporters all quietly collect data. Check what your error tracker sends when the verify endpoint throws an exception — a full request body containing the secret is exactly the kind of leak nobody notices for a year.

## 17 Sessions, cookies and step-up

Production baseline:

- HTTPS everywhere, no exceptions.
- Cookies set `Secure`, `HttpOnly`, `SameSite`.
- CSRF protection wherever you use cookie-based authentication.
- Session identifiers rotated on every privilege change — login, MFA completion, role change.
- No sensitive authentication material parked casually in localStorage.

For genuinely destructive actions, ask again even inside a valid admin session. This is step-up authentication, and it is where that stored `mfa_at` timestamp earns its place: if the last verification was more than a few minutes ago, challenge again.

```
Delete user
Change admin roles
Disable 2FA
Generate new recovery codes
Change password
Change payment settings
Delete production data

        |
        v
For security, verify your identity.
2FA Code: [ ______ ]    [ Continue ]
```

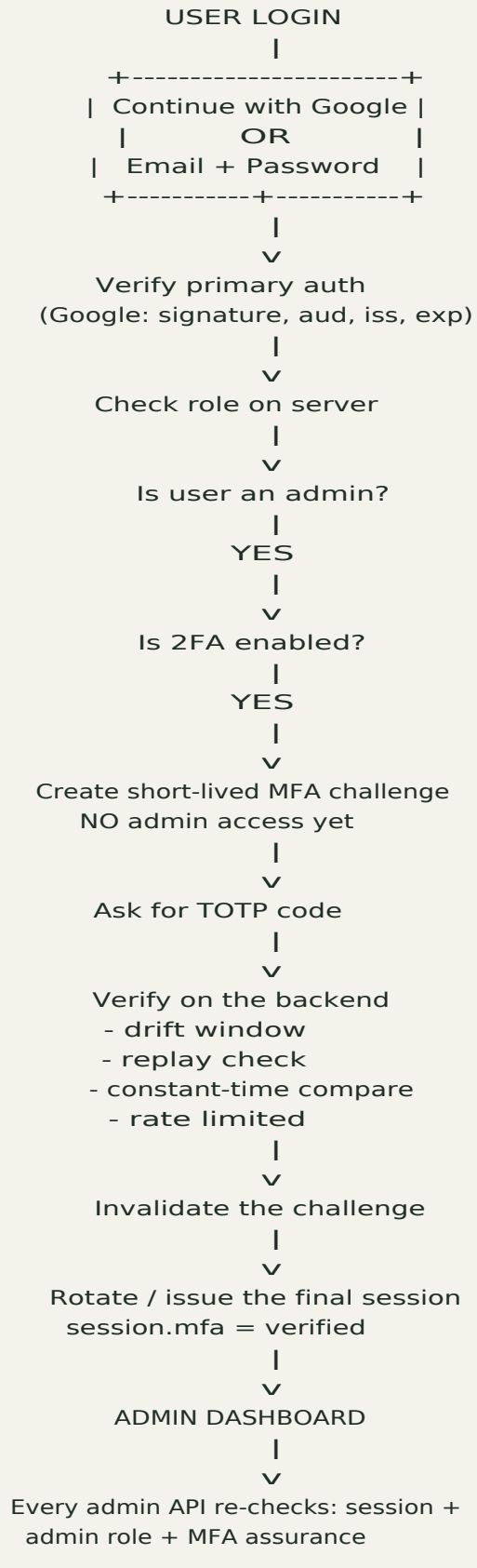
## 18 Keep an audit trail

Log the security events themselves, without ever logging the secrets. At minimum: 2FA enabled, 2FA disabled, recovery code used, repeated verification failures, and each successful admin login with time and IP. When something eventually looks wrong, this is the only record that tells you what actually happened.

REFERENCE

# The complete architecture

Every piece, in order, on one page.


USER LOGIN
Continue with Google
OR
Email + Password
Verify primary auth
(Google: signature, aud, iss, exp)
Check role on server
Is user an admin?
YES
Is 2FA enabled?
YES
Create short-lived MFA challenge
NO admin access yet
Ask for TOTP code
Verify on the backend
- drift window
- replay check
- constant-time compare
- rate limited
Invalidate the challenge
Rotate / issue the final session
session.mfa = verified
ADMIN DASHBOARD
Every admin API re-checks: session +
admin role + MFA assurance

BEFOREYOUSHIP

# Final security checklist

If you cannot tick a box, that box is a bug, not a preference.

### AUTHENTICATION CORE

- ☐ Authentication is verified on the server, never trusted from the client
- ☐ Admin authorisation is checked on the server
- ☐ Google ID tokens are validated for signature, `aud`, `iss` and `exp`
- ☐ Accounts are keyed on Google's `sub`, not on email

### THE TOTP SECRET

- ☐ Generated by the backend, cryptographically random
- ☐ Unique per admin, never shared or derived from guessable data
- ☐ Encrypted at rest (not hashed — the server needs it back)
- ☐ Encryption key lives outside the database: env secret, Secret Manager or KMS
- ☐ QR code generated locally — the `otpauth://` URI never touches a third-party service
- ☐ Manual-entry key shown alongside the QR
- ☐ First code verified before 2FA is switched on

### VERIFICATION

- ☐ A limited clock-drift window is accepted, and the server is NTP-synced
- ☐ Replay protection: a used time step is never accepted again
- ☐ Codes compared in constant time
- ☐ Strict attempt limits on TOTP verification
- ☐ Strict attempt limits on recovery-code verification too

### SESSIONS

- ☐ MFA challenge state is short-lived, single-use and bound to one login attempt
- ☐ The challenge cannot reach any normal or admin API
- ☐ A new session is issued or rotated after MFA succeeds
- ☐ MFA assurance is recorded server-side, not as a frontend flag
- ☐ Cookies use Secure, HttpOnly and SameSite; CSRF protection is in place

### EVERYTHING ELSE

- ☐ Every admin API independently enforces session, role and MFA assurance

- [ ] Recovery codes are random, single-use and stored hashed
- [ ] Disabling or changing 2FA requires re-authentication
- [ ] Sensitive actions can demand fresh verification (step-up)
- [ ] Secrets and OTPs never reach logs, analytics or error reporters
- [ ] Security events are audit-logged
- [ ] HTTPS everywhere

> **ONE HONEST LIMITATION**
>
> TOTP is a large improvement over passwords alone, but it is not phishing-proof. A convincing fake login page can capture a code and use it immediately — replay protection stops *reuse*, not that first real-time relay. For high-value admin systems, look at passkeys, WebAuthn and hardware security keys, which are phishing-resistant by design. OWASP recommends exactly this direction for privileged accounts.

Everything here is the architecture and the rules around it — deliberately not a “paste these twenty lines and you are secure” snippet, because the real code depends entirely on whether you are on Next.js, Firebase, Supabase, Django or Laravel. Take the checklist to whichever stack you use, and make it tick every box.

---

**Vikas Yadav**
Collabs and questions: **vikasbuilds.ofc@gmail.com**



## More things worth exploring

Find more free guides at vikasbuilds.com/resources