TECHNICAL APPENDIX.

The engineer's copy. Exact requests, reproduction scripts, what the code did wrong, and how it should have been built. Read the one-page Security Report first for the plain-language summary.

TARGET	ENDPOINT	SEVERITY	FIX TIME
[redacted]	[redacted]	Critical	[redacted]

Scope of this document

This is the technical companion to the [redacted] Security Report. It assumes you have read that one-pager. Below is everything an engineer needs to reproduce the finding, understand exactly what the code got wrong, and ship the fix. Every request here was run against the live endpoint using the account holder's own number, with no more than a handful of attempts.

Fastest path: hand this to your coding assistant

If you only want the app secured, start here. Paste the prompt below into an AI coding assistant (Claude Code, Cursor, Copilot) pointed at your backend repo. It states what the fix

has to achieve and leaves the how to the assistant, which should read the repo, match its stack and idioms, and pick the implementation. The rest of this document explains each layer, the reproduction, and why it works, for whoever reviews the change. Tune the numbers to your traffic before you ship.

```
Harden the OTP request endpoint POST
It currently sends an SMS with almost no limit, which lets attackers drain
the SMS budget, enumerate phone numbers, and lock users out. Read the repo
first, match its existing stack and conventions, and implement defense in
depth: check every limit below, reject if ANY is over budget, and keep all
counters in shared storage so every server agrees on the count.

1. When any limit is hit, respond 429 (not 500) with Retry-After,
   X-RateLimit-Limit, X-RateLimit-Remaining, and X-RateLimit-Reset.

2. Sliding-window rate limits on multiple keys, reject if ANY is over budget:
   - phone:   3 per 10 min, 10 per day
   - ip:      20 per hour
   - subnet:  /24 (IPv4) or /64 (IPv6), 200 per hour
   - device:  10 per hour   (device identity from step 4)
   - global:  circuit breaker, ~2000 SMS per minute

3. Treat the ip and subnet ceilings as a trigger to REQUIRE the step-4
   challenge, not a hard block, because carrier-grade NAT shares one IP
   among many real users. Keep phone and device as hard limits.

4. Require proof the caller is a genuine client BEFORE sending an SMS
   (for example, mobile app attestation, or an invisible web challenge).
   Respond 403 if that proof is missing or invalid.

5. Track the sent-vs-verified ratio per key. When requests are high and the
   verify ratio collapses, flag abuse and force the step-4 challenge.

6. Number hygiene: only accept country codes we serve, flag VoIP and
   disposable ranges, and cap velocity per country.

Rollout without locking out existing app versions: steps 1, 2, 3, 5, and 6
do not change the client contract, so apply them to the current endpoint
immediately (it is being actively exploited). Step 4 rejects any client that
cannot send the proof, so ship that on a new endpoint version (v2, e.g.
                                  or behind a client-capability flag, migrate
clients to it, then deprecate v1 once traffic has moved. Do not leave v1
without the rate limits in the meantime.

Do not change the existing success response shape. Add tests for each limit
and for the 429 headers. Explain any threshold you choose.
```

The short version

Your OTP endpoint at `POST` [redacted] does not rate limit correctly. It is the **only** way users log in, so a working attack here means users cannot get into your app.

We sent OTP requests across several phone numbers and delays. Sending requests with a 500ms gap succeeds **every single time**. Sending a burst all at once sometimes works and sometimes fails, depending on which backend server catches it. That inconsistency is itself a finding: your servers do not agree on the rate limit.

CRITICAL SEVERITY

This is not a deep architectural rewrite. It is a concrete rate-limiting bug with a clear, known fix. But because it sits on your login path, it needs to happen soon, before your next high-traffic event.

What we found

Issue 1 · Wrong HTTP status when rate limited

When we hit the limit, your API returns `500 Internal Server Error` instead of `429 Too Many Requests`.

This breaks client behaviour. A mobile app or web frontend that sees a 500 assumes the server crashed and retries immediately, which makes the flood worse. The correct answer to a 429 is "back off and wait." The correct answer to a 500 is "try again now." You are sending the wrong signal.

Issue 2 · No rate-limit headers

A rate-limited response should tell the client when it may retry. None of these are present:

HEADER	STATUS	PURPOSE
`Retry-After`	Missing	Seconds until retry is allowed
`X-RateLimit-Limit`	Missing	Max requests in the window
`X-RateLimit-Remaining`	Missing	Requests left before block
`X-RateLimit-Reset`	Missing	When the window resets

Issue 3 · Rate limiting is per phone number, not per IP

Your backend counts OTP requests by phone number. It does not count by IP address or session. So an attacker just changes the phone number to get a fresh limit every time.

REPRODUCE

```
curl -X POST                                                                \
  -H "Content-Type: application/json" \
  -d
# 200 OK

curl -X POST                                                                \
  -H "Content-Type: application/json" \
  -d
# 200 OK  (different number, fresh limit)
```

Loop through [redacted] to [redacted] and each number lets you send at least one OTP before its own limit kicks in. There is no IP-level ceiling stopping the loop.

Issue 4 · The threshold itself is too lenient

We measured how much traffic the endpoint actually allows. The results are in the next section.

What an attacker can do

1 Enumerate the phone numbers in your database

Each number is independent, so you can test thousands and learn which ones have accounts.

```
#!/bin/bash
for i in {0000..5000}; do
  curl -s -X POST
    -H "Content-Type: application/json" \
    -d                                  > /dev/null
  sleep 0.1
done
```

After this run, an attacker holds the phone numbers of up to 5,000 of your users. That list feeds phishing, spam, harassment, or a competitor's cold-outreach.

2 Drain your SMS budget

Every OTP costs you money at your SMS provider. Spam requests at popular numbers and the meter runs.

```
# Rapid-fire requests at a single number
for i in {1..100}; do
  curl -s -X POST
    -H "Content-Type: application/json" \
    -d                                  &
done
wait
```

If you carry a 10,000-message monthly quota and an attacker burns it during a tournament, you are out of budget and paying customers cannot log in. The financial hit and the outage arrive together.

3 Lock users out during an event

Run attack 2 during your Friday-afternoon tournament. Users cannot verify their phones, cannot get in, and flood your support. Paying customers churn to a competitor. This is the scenario that turns a quiet bug into a bad day.

How we reproduced it

Test 1 · Sequential requests, 500ms apart

```
const phone =

for (let i = 1; i ⩽ 10; i++) {
  const res = await fetch(

    { method: 'POST',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify({ phone_number: phone }) }
  );
  console.log(`Request ${i}: ${res.status}`);
  await new Promise(r ⇒ setTimeout(r, 500));
}
```

- **10 / 10 succeeded (200 OK).** The limit resets inside 500ms. An attacker just adds a half-second sleep and sends forever.

Test 2 · All 10 at once, no delay

```
const phone =
const reqs = Array.from({ length: 10 }, () ⇒

    { method: 'POST',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify({ phone_number: phone }) })
);
const results = await Promise.all(reqs);
results.forEach((r, i) ⇒ console.log(`Request ${i+1}: ${r.status}`));
```

- **2 / 10 succeeded, 8 returned 500.** Some backend servers enforce a limit, some do not, or they do not share state. Spread the same burst across different numbers and all of them get through.

Verify it yourself

Check 1 · Send one request

```
curl -X POST                                                                    \
  -H "Content-Type: application/json" \
  -d
```

Expected: `200` with `{"message":"OTP sent successfully to your SMS. Valid for 10 minutes."}`

Check 2 · Look for rate-limit headers

```
curl -i -X POST
  -H "Content-Type: application/json" \
  -d                                    | head -20
```

You will not find `Retry-After`, `X-RateLimit-Limit`, `X-RateLimit-Remaining`, or `X-RateLimit-Reset` anywhere in the response.

Check 3 · Different numbers, fast

```
for i in {1..5}; do
  echo "Phone
  curl -s -X POST
    -H "Content-Type: application/json" \
    -d
  echo ""
done
```

All five succeed. Different numbers, different limits. That proves the limit is per-number, not per-IP.

How to fix it

One clarification first, because it is the part most teams get wrong: **there is no single key that fixes this.** "Just rate-limit by IP" is not the answer. An IP is cheap to rotate (a residential proxy pool is a few dollars an hour), and worse, carrier-grade NAT puts thousands of real, unrelated users behind one address, so a tight per-IP limit blocks genuine customers while a determined attacker rotates around it. The fix is defense in depth: limit on several keys at once, prove the caller is really your app, and watch the numbers that only an attacker moves.

> HOW THE BIG PLATFORMS DO IT
>
> Firebase Phone Auth, Auth0, and Twilio Verify all layer the same three things: multi-dimension rate limits (number, IP, subnet, device, global), a proof-of-genuine-client step (reCAPTCHA / App Check / Play Integrity), and abuse analytics on the request-to-verify ratio. None of them rely on IP alone. The recommendations below are that model, scoped to your stack.

Fix 1 · Return 429 with proper headers

Swap the 500 for a 429 and attach the standard headers so clients back off correctly. Django example:

```
from django.http import JsonResponse
from django.core.cache import cache
import time

def request_otp(request):
    phone = request.POST.get('phone_number')
    key = f"otp_rate:{phone}"
    attempts = cache.get(key, [])

    if len(attempts) ⩾ 3:
        reset = int(attempts[0]) + 600          # 10-minute window
        return JsonResponse(
            {"error": "Too many requests"},
            status=429,
            headers={
                "Retry-After": str(max(0, reset - int(time.time()))),
                "X-RateLimit-Limit": "3",
                "X-RateLimit-Remaining": "0",
                "X-RateLimit-Reset": str(reset),
            },
        )

    send_sms(phone, generate_otp())
    attempts.append(int(time.time()))
    cache.set(key, attempts, 600)
    return JsonResponse({"message": "OTP sent successfully"})
```

Fix 2 · Limit on several keys at once, most restrictive wins

Check the request against every key it belongs to and reject if *any* is over budget. Each key catches a different attack shape, so an attacker has to beat all of them at the same time. Use a sliding window per key.

KEY	BUDGET (STARTING POINT)	CATCHES
`phone`	3 / 10 min, 10 / day	Spamming one victim's number
`ip`	20 / hour, escalate before block	One host looping many numbers
`ip_subnet` (/24 v4, /64 v6)	200 / hour	An attacker cycling IPs in one block
`device` (see Fix 3)	10 / hour	One app install or browser, any IP
`global`	circuit breaker on total SMS/min	Caps blast radius and spend, whatever the source

```
def request_otp(request):
    phone  = request.POST['phone_number']
    ip     = get_client_ip(request)
    subnet = ip_subnet(ip)                 # /24 for v4, /64 for v6
    device = device_id(request)            # from Fix 3

    keys = [
        ("phone",  f"otp:phone:{phone}",   3,   600),
        ("ip",     f"otp:ip:{ip}",         20,  3600),
        ("subnet", f"otp:net:{subnet}",    200, 3600),
        ("device", f"otp:dev:{device}",    10,  3600),
        ("global", "otp:global",           2000, 60),
    ]
    for name, key, limit, window in keys:
        if sliding_window_count(key, window) ⩾ limit:
            return rate_limit_error(name)   # 429 + headers from Fix 1

    send_sms(phone, generate_otp())
    for _, key, _, window in keys:
        sliding_window_incr(key, window)
    return JsonResponse({"message": "OTP sent successfully"})
```

> **Do not hard-block the IP and subnet keys. Escalate them.**
>
> Because CGNAT shares an IP across many real users, treat the `ip` and `subnet` ceilings as a trigger for *friction*, not a wall: over budget means "now require the Fix 3 challenge", and only block after that also fails. The `phone` and `device` keys can stay hard limits. This is how you stop the attacker without locking out a whole mobile carrier's worth of customers.

Fix 3 · Prove the caller is your real app, not a script

Rate limits raise the cost of the attack. They do not stop it, because `curl` can send a perfectly valid request. What actually stops a script is making the endpoint refuse anything that cannot prove it is a genuine, untampered client. You are mobile-first, so this is your highest-leverage fix.

- **Mobile app:** require an attestation token on every OTP request. **Apple App Attest** (iOS) and **Google Play Integrity** (Android) hand your app a signed token that your server verifies before sending SMS. A script, an emulator, or a repackaged APK cannot mint a valid one. This single control kills the curl-based flood outright.
- **Web:** require a **Cloudflare Turnstile** or **reCAPTCHA v3** token, verified server-side, before the SMS goes out. Invisible to real users, hard for automation.

```
# Reject before spending an SMS if the client can't attest
token = request.headers.get("X-App-Attest")   # or Turnstile token on web
if not verify_attestation(token, device_id(request)):
    return JsonResponse({"error": "Unverified client"}, status=403)
```

> **Ship this on a v2 endpoint so old app versions don't break**
>
> This is the one control that changes the client contract: an app version that cannot send a token gets rejected. So make it mandatory on a new endpoint version ([redacted]) or behind a client-capability flag, move clients to it, then deprecate v1 once traffic has shifted. The rate limits in Fixes 1, 2, and 5 do not change the contract, so put those on v1 right away. Do not wait on the v2 migration to stop the active abuse.

Fix 4 · Watch the request-to-verify ratio

This is the cheapest strong signal you have, and it is exactly what SMS-pumping fraud trips. A real user verifies most codes they ask for. An attacker draining your budget requests

thousands and verifies almost none. Track sent-versus-verified per IP, per subnet, per device, and per number prefix, and act when the ratio collapses.

```
# Per key, keep a rolling (sent, verified) count.
# If sent is high and verified/sent < ~0.1, auto-escalate to the
# Fix 3 challenge, alert, and rate-limit that key harder.
if sent > 50 and verified / max(sent, 1) < 0.10:
    flag_abuse(key); require_challenge(key)
```

Pair this with basic number hygiene: only accept country codes you actually serve, flag known VoIP and disposable ranges, and cap velocity per country. SMS-pumping fraud targets specific carrier ranges to earn payouts, so geo limits cut the economics out from under it. Twilio exposes this as Verify Fraud Guard and geo-permissions if you are on their stack.

Fix 5 · Share the limit state across all backend servers

Test 2 showed your servers disagree on the count, which means the limit lives in per-process memory. Move every counter above into Redis so all servers read and write one shared state.

```
# settings.py
CACHES = {
    "default": {
        "BACKEND": "django_redis.cache.RedisCache",
        "LOCATION": "redis://127.0.0.1:6379/1",
        "OPTIONS": {"CLIENT_CLASS": "django_redis.client.DefaultClient"},
    }
}
```

Priority and sequencing

Ship it in layers. Each row is independently deployable and each one raises the cost of the attack.

STEP	WHAT IT CLOSES	EFFORT
1. 429 + headers (Fix 1)	Wrong status code, clients retry-storming	2–3 hrs
2. Multi-key limits in Redis (Fix 2 + 5)	Single-key limiting, server disagreement, the lenient threshold	1–2 days
3. App attestation / Turnstile (Fix 3)	Scripted floods and enumeration outright	2–4 days
4. Verify-ratio analytics + geo (Fix 4)	SMS-pumping, slow distributed abuse	2–3 days

Steps 1 and 2 stop the bleeding and can land inside a day or two. Step 3 is the one that actually ends the attack class rather than slowing it, and it is worth doing next. Step 4 is the long-term guardrail that keeps cost and abuse visible.

Two things we noticed on the way

Your landing page is mobile-only

Great for users on phones mid-tournament. But investors, partners, and desktop visitors land on a layout built for small screens. A responsive desktop view opens the app to the audiences who decide whether to back you or partner with you.

The server disagreement points deeper

Backend servers that don't agree on a rate limit often don't agree on other shared state either: sessions, caches, counters. Worth a full architecture pass before you scale past 100k users, so the next event doesn't surface this class of bug live.

Next step

Engagement options and how to reach us are in the one-page Security Report that accompanies this appendix. To start, email **hello@gattyworks.com** and we'll take it from there.