

preflight

A pre-launch leak check for solo-built (often AI-assisted) apps.

It reads a project folder and reports the handful of mistakes that most often expose real user data after launch: secrets that look committed, a service-role key in browser code, Postgres tables with no row-level security, and public storage buckets.

No network. No dependencies (Python 3, stdlib only). Nothing is uploaded; it only reads files.

Run

```
python3 preflight.py [project_dir]
```

Exit code is `1` if anything HIGH was found, `0` otherwise, so it drops into a pre-commit hook or CI step.

What it flags

- **.env present but not in .gitignore** — keys can end up in repo history.
- **Hardcoded secrets** — `service_role`, `sk-...`, JWTs, AWS/Google keys.

HIGH when the file looks like client/browser code, MED otherwise.

- **Public env prefixes holding secrets** — `NEXT_PUBLIC_..._SECRET`, `VITE_...`.
- **Postgres tables with no row-level security** — a table reachable through

Supabase's API is readable by anyone with the public anon key unless RLS is on.

- **No policies anywhere in the migrations** — RLS with no policy denies everything; RLS off means the table is wide open.
- **Public storage buckets** — every file served to anyone with the URL.

What it does not do

It is a static read of your files, not a live scan of your database. It cannot see your deployed project's actual policies. Treat it as the checklist you run before you push, and verify the real thing in the Supabase dashboard.

The rule behind it

Your anon key is public by design. Everything it can reach is public. RLS plus a policy per table is what makes the difference between a private app and a public data dump.

Full source

```
#!/usr/bin/env python3
"""
```

preflight - a pre-launch leak check for solo-built (often AI-assisted) apps.

It reads a project folder and reports the handful of mistakes that most often expose real user data after launch: secrets that look committed, a service-role key in browser code, Postgres tables with no row-level security, and public storage buckets.

No network. No dependencies. Nothing is uploaded. It only reads files.

```
python3 preflight.py [project_dir]
```

Exit code is 1 if anything HIGH was found, else 0, so you can wire it into CI.

```
"""

import argparse
import os
import re
import sys

SKIP_DIRS = {
    ".git", "node_modules", ".next", "dist", "build", "out", ".venv", "venv",
    "__pycache__", ".cache", "coverage", ".turbo", ".svelte-kit", "vendor",
}
TEXT_EXT = {
    ".py", ".js", ".jsx", ".ts", ".tsx", ".mjs", ".cjs", ".vue", ".svelte",
    ".html", ".css", ".scss", ".sql", ".env", ".sh", ".yaml", ".yml", ".toml",
    ".rb", ".go", ".php", ".java", ".kt", ".cs", ".swift", ".json", ".md",
}
MAX_BYTES = 1_000_000
CLIENT_HINT = re.compile(
    r"^(^|/)(src|app|pages|components|public|static|client|frontend|web)(/|$)", re.I
)
ENV_FILES = (".env", ".env.local", ".env.production", ".env.development",
             ".env.staging", ".env.prod")
SECRETISH = [
    (re.compile(r"service_role", re.I), "a Supabase service-role key"),
    (re.compile(r"\bsk-[A-Za-z0-9]{16,}\b"), "an API secret key (sk-...)"),
    (re.compile(r"\beyJ[A-Za-z0-9_-]{10,}\.[A-Za-z0-9_-]{10,}\.[A-Za-z0-9_-]{10,}\b"),
     "a JWT (signing key or service token)"),
    (re.compile(r"\b(AKIA[0-9A-Z]{16})\b"), "an AWS access key id"),
    (re.compile(r"\bAIza[0-9A-Za-z_-]{35}\b"), "a Google API key"),
]
PUBLIC_ENV = re.compile(r"\b(NEXT_PUBLIC_|VITE_|PUBLIC_|REACT_APP_)[A-Z0-9_]+", )
CREATE_TABLE = re.compile(
    r"create\s+table\s+(?:if\s+not\s+exists\s+)?\[''\]?([a-zA-Z0-9_\.\'']+)[\'\']?",
    re.I)
ENABLE_RLS = re.compile(
    r"alter\s+table\s+\[''\]?([a-zA-Z0-9_\.\'']+)[\'\']?s+enable\s+row\s+level\s+security",
    re.I)
PUBLIC_BUCKET = re.compile(
    r"(public\s*[:=]\s*true|create\s+bucket\s*\^[^)]*public\s*=\s*true)", re.I)

FINDINGS = []

def add(sev, path, line, title, why, fix):
    FINDINGS.append({
        "sev": sev, "path": path or "-", "line": line,
        "title": title, "why": why, "fix": fix,
    })
```

```

def read_text(path):
    try:
        with open(path, "r", encoding="utf-8", errors="ignore") as fh:
            return fh.read(MAX_BYTES)
    except OSError:
        return None

def walk(root):
    for dirpath, dirnames, filenames in os.walk(root):
        dirnames[:] = [d for d in dirnames if d not in SKIP_DIRS]
        for name in filenames:
            yield os.path.join(dirpath, name)

def rel(root, path):
    return os.path.relpath(path, root).replace(os.sep, "/")

def looks_like_client(relpath):
    return bool(CLIENT_HINT.search(relpath))

def parse_gitignore(root):
    gi = os.path.join(root, ".gitignore")
    text = read_text(gi) or ""
    patterns = [ln.strip() for ln in text.splitlines()
                 if ln.strip() and not ln.startswith("#")]
    return patterns

def env_ignored(env_name, patterns):
    for pat in patterns:
        core = pat.lstrip("!").rstrip("/")
        if core in (env_name, "*" + env_name) or core.startswith(env_name):
            return True
        if core in ("*.env", ".env*", "*.local") and env_name.startswith(".env"):
            return True
        if core == "*" and not pat.startswith("!"):
            return True
    return False

def check_git_and_env(root, patterns):
    for name in ENV_FILES:
        p = os.path.join(root, name)
        if os.path.isfile(p) and not env_ignored(name, patterns):
            add("HIGH", rel(root, p), 1,
                "env file not ignored by git",
                f"{name} is present and .gitignore does not clearly exclude it, "
                "so keys in it can end up in the repo history.",
                f"add `{name}` to .gitignore and rotate any key it holds")

def check_secrets(root):
    for path in walk(root):
        name = os.path.basename(path)
        ext = os.path.splitext(name)[1].lower()
        if ext not in TEXT_EXT and not name.startswith(".env"):
            continue
        r = rel(root, path)

```

```

# skip the scanner itself
if r.endswith("preflight.py"):
    continue
text = read_text(path)
if text is None:
    continue
lines = text.splitlines()
for i, line in enumerate(lines, 1):
    if line.strip().startswith(("#", "//", "*")):
        continue
    for rx, label in SECRETISH:
        if rx.search(line):
            in_client = looks_like_client(r)
            sev = "HIGH" if in_client else "MED"
            add(sev, r, i,
                f"possible hardcoded secret ({label})",
                ("this file looks like browser/client code, so anything in it "
                 "ships to users" if in_client else
                 "a literal secret in source tends to be committed and shared"),
                "move it to an environment variable and rotate the key")
            break # one secret per line is enough
    if PUBLIC_ENV.search(line) and ("secret" in line.lower()
                                     or "service" in line.lower()):
        add("HIGH", r, i,
            "public env var looks like it holds a secret",
            "client-exposed prefixes (NEXT_PUBLIC_/VITE_/etc.) are bundled "
            "into the browser; a secret there is public.",
            "never put a service-role or secret key behind a public prefix")

def check_rls(root):
    for path in walk(root):
        if not path.lower().endswith(".sql"):
            continue
        r = rel(root, path)
        text = read_text(path) or ""
        created = {m.group(1).strip('').lower() for m in CREATE_TABLE.finditer(text)}
        if not created:
            continue
        secured = {m.group(1).strip('').lower() for m in ENABLE_RLS.finditer(text)}
        # also accept RLS enabled in any other migration in the project
        project_secured = set(secured)
        for other in walk(root):
            if other.lower().endswith(".sql") and other != path:
                ot = read_text(other) or ""
                project_secured |= {m.group(1).strip('').lower()
                                   for m in ENABLE_RLS.finditer(ot)}
        for table in sorted(created):
            if table in project_secured:
                continue
            add("HIGH", r, 1,
                f"table `{table}` has no row-level security",
                "Postgres tables reachable through Supabase's API are readable by "
                "anyone holding the public anon key unless RLS is enabled and "
                "policies restrict rows.",
                f"`alter table {table} enable row level security;` plus a policy "
                "per operation you allow")
        if "create policy" not in text.lower():
            project_policies = False
            for other in walk(root):
                if other.lower().endswith(".sql"):
                    if "create policy" in (read_text(other) or "").lower():
                        project_policies = True

```

```

        break
    if not project_policies:
        add("MED", r, 1,
            "no create policy anywhere in the migrations",
            "RLS with no policy denies everything; but if RLS is off, the "
            "table is wide open. Policies are where you say who may "
            "read/write.",
            "add one policy per table and operation you intend to allow")
    for m in re.finditer(r"create\s+bucket\s*\([^)]*\)", text, re.I):
        if "public" in m.group(1).lower() and "true" in m.group(1).lower():
            add("MED", r, 1,
                "storage bucket created public",
                "a public bucket serves every file to anyone with the URL.",
                "use a private bucket and signed URLs unless the files are "
                "genuinely non-sensitive")
    if PUBLIC_BUCKET.search(text) and "storage" in text.lower():
        pass

def main():
    ap = argparse.ArgumentParser(description="pre-launch leak check")
    ap.add_argument("path", nargs="?", default=".", help="project folder")
    args = ap.parse_args()
    root = os.path.abspath(args.path)
    if not os.path.isdir(root):
        print(f"not a folder: {root}")
        return 2

    patterns = parse_gitignore(root)
    check_git_and_env(root, patterns)
    check_secrets(root)
    check_rls(root)

    order = {"HIGH": 0, "MED": 1, "LOW": 2}
    FINDINGS.sort(key=lambda f: (order.get(f["sev"], 9), f["path"]))
    highs = sum(1 for f in FINDINGS if f["sev"] == "HIGH")

    print(f"preflight: {root}")
    print(f"{len(FINDINGS)} finding(s), {highs} high\n")
    if not FINDINGS:
        print("nothing obvious. still rotate keys that were ever in a repo.")
        return 0
    for f in FINDINGS:
        print(f"[{f['sev']}] {f['path']}:{f['line']} {f['title']}")
        print(f"    why: {f['why']}")
        print(f"    fix: {f['fix']}\n")
    print("high findings are the ones to fix before launch.")
    return 1 if highs else 0

if __name__ == "__main__":
    sys.exit(main())

```