iVerify.

iverify.com

A FORENSIC DEEP DIVE INTO ANDROID INTRUSION LOGGING

Co-Authors:

David Gillies,
Head of Android Research, iVerify

Lorena Carthy-Wilmot,
Head of Security Strategy (Europe), iVerify

TABLE OF CONTENTS

Introduction

Until now, investigating a compromised Android device has meant working with logs that were never designed to survive an attack. Intrusion Logging (IL) changes that. Introduced in Android 16 as part of Android's Advanced Protection Mode (AAPM), IL records the device events most relevant to spotting a compromise, encrypts them with keys only the user controls, and stores them off the device where an attacker cannot reach them. Google built the feature with Amnesty International's Security Lab, which served as a design partner throughout AAPM's development.

Amnesty International's Security Lab described this as a major step forward for forensic work in its technical briefing,[1] and our testing bears that out. There are two practical catches, though: Google is rolling the feature out in stages, and logging only begins once the user has turned AAPM on, so the protection has to be in place prior to any compromise. That timing matters more on Android than it might elsewhere, because the platform's own logs are transient by design.

The inherent difficulty of mobile forensics stems from structural limitations in how an OS maintains logs, a vulnerability that prominent spyware providers exploit using aggressive anti-forensic techniques. This has been seen in Pegasus, Graphite, and Predator on Android devices.

IL marks the first instance of a major manufacturer releasing a feature dedicated to bridging this forensic gap. Built specifically for consensual analysis, it allows users to opt in proactively. Should a compromise be suspected later, they can provide a reliable, tamper-resistant record to their chosen analyst. Moving beyond basic design specs, iVerify's research examines actual Intrusion Logging data from live devices to see what is recorded, how the data is organized, and its utility in detecting breaches, linking network traffic to specific applications, mapping attacker movements, and validating other investigative findings, while also identifying current limitations and manual parsing challenges.

IL is currently in a limited release. It requires a modern Pixel device on Android 16 or later with account-level Advanced Protection enabled. Due to a phased rollout, even compatible hardware may not yet have access. Crucially, as mentioned, IL lacks retroactivity; it only records data from the point of activation. Therefore, it serves as a preemptive tool for those anticipating targeted attacks rather than a retroactive safety net.

1 Amnesty International's Security Lab [May 12th, 2026]

Intrusion Logging in Detail

Intrusion logging is the first mainstream mobile feature genuinely built for the consensual-forensics workflow rather than adapted to it, and at a technical level, it is best understood as a curated, tamper-resistant, end-to-end encrypted audit trail.

For forensic analysts, these properties are vital because the log provides a specific, security focused record of events, including authentication attempts, DNS and connection activity, process starts, and package installations. It also tracks critical system changes like certificate authority modifications, ADB sessions, boot states, and AAPM policy updates.

A key security advantage is that these logs are never written to the device as world-readable files. They can only be reached through the same settings menu that activates them, and they are backed up to the owner's Google Account using end-to-end encryption, meaning not even Google can read them.[2] Because an attacker cannot silently read, modify, or delete entries, the record stays trustworthy and under the user's sole control.

Furthermore, as the encrypted archives are stored at the account level, the audit trail persists even if on-device data is lost or wiped, providing a forensic history that survives the typical cleanup operations performed by common sophisticated spyware.

How IL Differs from Logcat and the Classic Bugreport

It is worth providing detail about why IL is a step change rather than a repackaging of existing logs. The everyday Android logging surfaces an analyst would otherwise reach for were mainly built for engineering, not forensic analysis or investigation:

Source	What it was built for	Forensic limitation
Logcat	App/system debugging	Small fixed circular buffers; overwritten in minutes–hours
Debuggerd / tombstones	Crash diagnostics	Limited, auto-rotated
Bug Report	Support and troubleshooting	Point-in-time snapshot, thin on security context, typically only useful for the last 1 to 2 days
Intrusion Logging	Security and intrusion detection	Purpose-built, curated, encrypted, retained far longer

2 Advanced Protection Program [2026]

For us at iVerify, the retention difference is the headline. Our research and work shows that while a bug report may only give you a day or two of context, Intrusion Logging, when compared to traditional Android logging mechanisms, significantly extends that retention window to months instead of days.

Google have stated that encrypted logs are stored on their servers for 12-months,[3] during which manual deletion is not possible by either the user or Google. Regarding privacy, anyone with both the Google Account and the device screen lock could potentially retrieve these logs, this could potentially include law enforcement with physical access to the device and a legal warrant.

Analysts coming from Apple's ecosystem will reach for sysdiagnose archives and Unified Logs as rich log sources. Those are powerful, but still general-purpose diagnostic exports repurposed for security work. Additionally, they are collected at a point in time and not designed as a durable, consent-driven intrusion record. Android's Intrusion Logging inverts that model; rather than mining diagnostics that happen to contain useful traces, it produces a dedicated, retained, user-owned security log.

Under the Hood: How Intrusion Logging Works

It is also worth understanding how Intrusion Logging produces its data, because the design explains both its strengths and its limits. IL is not an ordinary app observing Android from the outside: its collection pipeline is implemented inside Android's `system_server`. Android exposes the framework service `android.security.intrusiondetection.IIntrusionDetectionService`, which controls collection and aggregation, while the separate Advanced Protection service provides the wider protection mode in which the feature is offered. When Intrusion Logging is enabled, the service activates its security-log and network-log data sources.

Security events, including process launches, ADB activity, certificate changes, keyguard events and boot-related records are delivered through Android's `android.app.admin.SecurityLog` infrastructure: some originate in native or kernel-facing audit mechanisms, while others are written directly by framework services. The same SecurityLog family has historically supported enterprise device auditing, but Intrusion Logging enables Android's newer internal audit path for a consenting individual user without requiring a Device Owner or other enterprise management.

DNS and outbound connection events are obtained through Android's netd connectivity-event pipeline. The callback supplies the destination information and originating UID; the framework then resolves that UID to an installed package name. This is why exported DNS and connection records can be attributed to the application responsible for the activity.

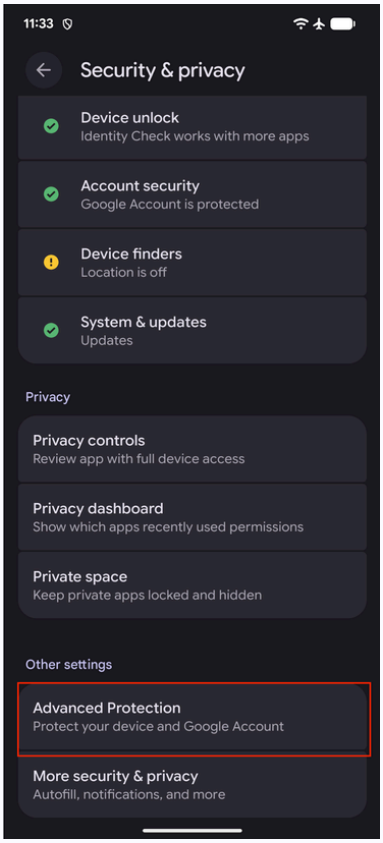
The framework passes collected events over a local Binder connection to the explicitly configured transport service in Google Play services, `com.google.android.gms.intrusiondetection.service.cIntrusionDetectionEventTransportService`. On the tested device this connection was opened by `system_server`, running as UID 1000. The receiving service must be declared with Android's signature-level `BIND_INTRUSION_DETECTION_EVENT_TRANSPORT_SERVICE` permission, preventing an untrusted process from binding to it, while the framework's explicit component configuration ensures that events are directed to the intended GMS service.

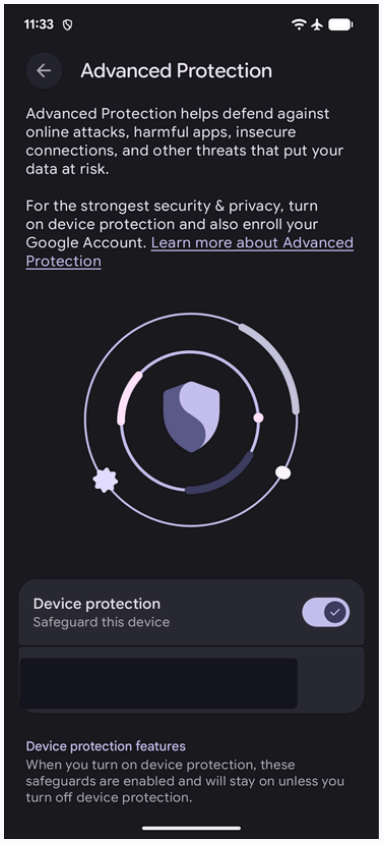
3 Log your Android device activity with Advanced Protection [2026]

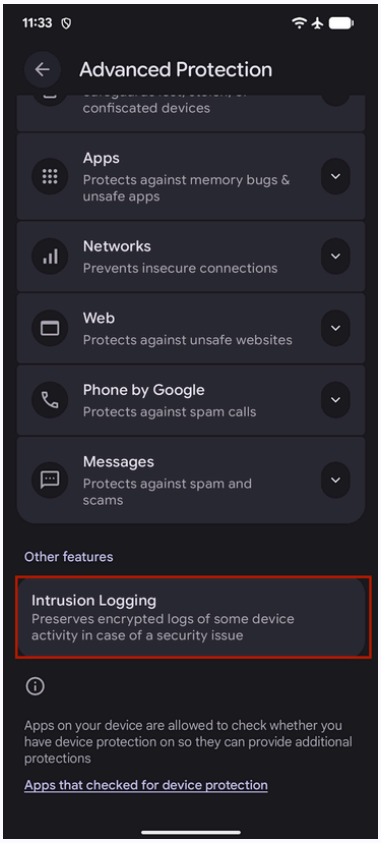
For defenders, one practical consequence is that you can confirm Intrusion Logging is running on a device without being able to read what it recorded. A Bug Report or live inspection will show the intrusion detection and advanced protection system services present, however it will not show the telemetry itself, which is encrypted before it is stored.

Pulling the Data

On Pixel devices, the Intrusion Logging menu can be found at:
`Settings -> Security & privacy -> Advanced Protection -> Intrusion Logging`

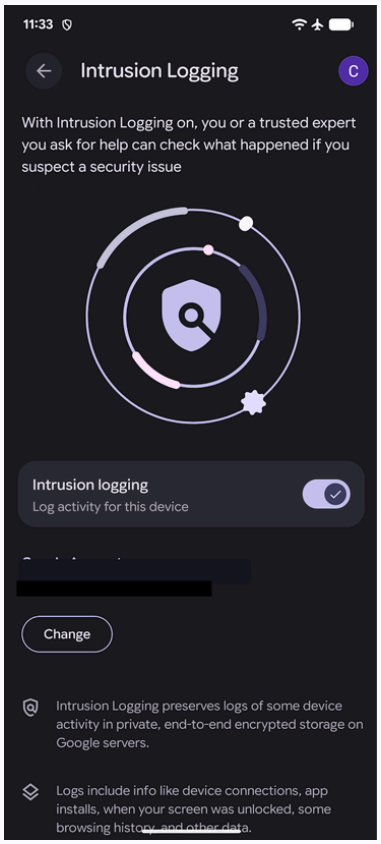


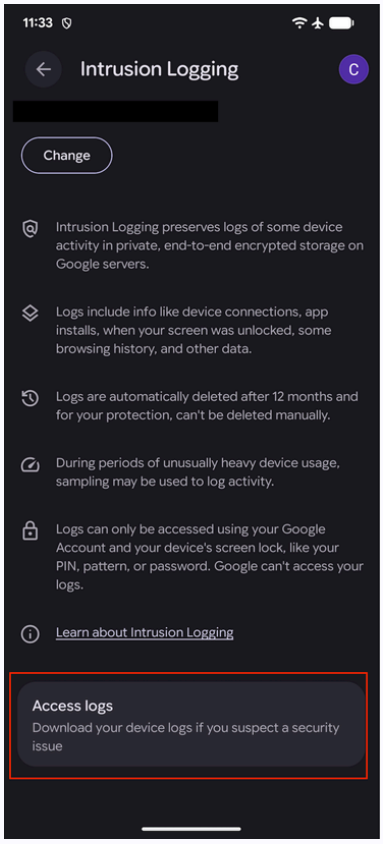


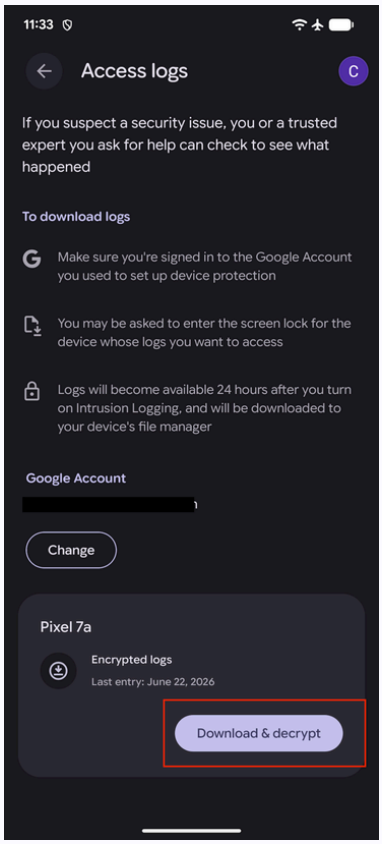


In our testing, the visible prerequisites were Android 16 or later, device-level Advanced Protection, and account-level Google Advanced Protection enrollment. At time of writing, Intrusion Logging is controlled by a server-side Google flag, so two accounts on similar hardware may have different results in terms of availability of this feature. Once the feature is available, files can be retrieved using the following steps:

1. Open Intrusion Logging.
2. Tap Access logs.
3. Choose the device.
4. Tap Download & decrypt.
5. Authenticate with the device screen lock.
6. Retrieve the downloaded bundle from the Downloads folder or pull over ADB.







Logs are automatically deleted after 12 months, cannot be deleted manually, and may be sampled during periods of unusually heavy device usage. A Twelve-month retention gives investigators a much longer runway than many traditional Android artifacts, and the inability to manually delete logs without full device control and user credentials is important for tamper resistance.

Download & decrypt does not give you the live state of the phone, it provides the latest cloud-backed-up bundle, which may not include data for the last 12-24 hours. We saw this clearly across repeated pulls from a Pixel 10 Pro device, in our testing a collection triggered at 22:00 UTC on June 8 had a dataset which ended at 22:39 UTC on June 7. Subsequent collections resumed exactly where the first one stopped, re-delivered the earlier files byte-for-byte, and added the next backed-up window.

For incident response, it is worth noting that the most relevant activity may be the most recent activity, and that may not have reached the cloud-backed-up bundle yet, and that the first set of logs only become available 24 hours after Intrusion Logging is first enabled.

After the on-device decrypt step, Android writes the bundle to the following path:
`/sdcard/Download/Intrusion Logging/`

On the device this appears as a ZIP file, for example:
`Pixel 7a_20260623.zip`

The filename date is not a trustworthy event date, it reflects the backup or upload window, not necessarily the day the activity occurred.

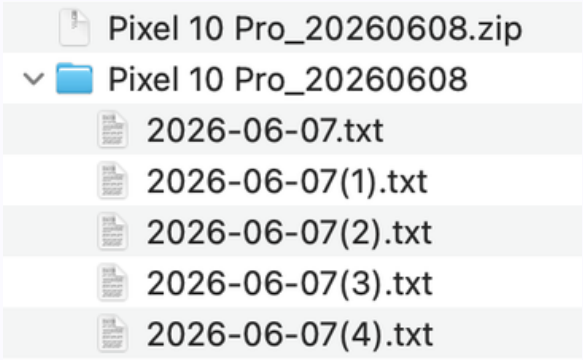
In our workflow, we pulled the files from the IL directory over ADB and extracted the ZIP locally. The extracted contents are plaintext JSONL (delivered as `.txt` files). No additional decryption step is required once the user has approved the download on the device.

For incident response, one capability stands out: logs can be retrieved from one device onto another. Intrusion Logging binds to a single Google Account per device, but that account can be changed, and this is what underpins Google's *trusted expert* model. In our testing, we signed a colleague's Google Account into a spare Pixel, switched Intrusion Logging to their account, and their device immediately appeared in the Access logs screen with its full encrypted history ready to download. Because the archive lives with the account rather than the phone, this works even if the original device has been powered off, wiped, or seized.

Enrolling a new device into an account's logs prompted us for the screen lock of the original device, the one whose logs we wanted, which is the step that unwraps the account's encryption key onto the new hardware. Only after that did routine downloads fall back to the local device's screen lock. So an analyst needs two things from the person they are helping: their Google Account sign-in, which under Advanced Protection means a passkey or security key, and that person's device screen lock.

Anatomy of an IL Bundle

Extracting the bundle gives a directory of JSON Lines files with .txt extension



As mentioned in the previous section, the filename date is not a trustworthy event date, it reflects the backup or upload window, not necessarily the day the activity occurred. We saw files named for June 9 containing events from June 7 through June 9, and the numbered suffixes are not a reliable chronology either, any parser or analyst should sort by `event_time`,not filename.

Each file roll held up to 8,192 events, and our early pulls each contained roughly 93,000 to 121,000 events (13 to 20 MB extracted). As retrieval re-delivers previously backed-up files byte-for-byte, an incremental pull one day later added only around 28,000 new events. The largest retrieval we analyzed contained 596,833 unique events across 83 files and covered activity spanning 15 days.

Every event we observed had one of three top-level types:

Event type	What it records	Why it matters
`dns_event`	Hostname lookups, resolved IPs, requesting package	Per-app DNS attribution for infrastructure and C2 hunting
`connect_event`	Destination IP, port, requesting package	Network connection metadata, including raw-IP connections
`security_event`	Android security events wrapped as nested subtypes	Unlocks, process launches, ADB, installs, boot state, key events, policy changes

The DNS event type is especially useful because it preserves context. Each lookup is recorded not only as a hostname, but also contains the package which made the request.


```
{
  "dns_event": {
    "event_time": 1783505607772,
    "package_name": "com.whatsapp",
    "hostname": "g.whatsapp.net",
    "ip_addresses": ["/57.144.239.33"],
    "ip_addresses_count": 1
  }
}
```


Connection events are narrower but still valuable: they show the package, destination IP, and port, though not byte counts, direction, or protocol. They also line up with the DNS stream. A few minutes before the connection below, WhatsApp resolved g.whatsapp.net to this same address, 57.144.239.33, and here it connects to that address on port 5222, WhatsApp's messaging port. Because every event is tagged with the app responsible, you can follow a single application from lookup to connection.


```
{
  "connect_event": {
    "event_time": 1783505792694,
    "package_name": "com.whatsapp",
    "port": 5222,
    "ip_address": "/57.144.239.33"
  }
}
```


Of the three event types, `security_event` is where most of the forensic value lives. Structurally, it is just a wrapper, an `event_id` and an `event_time` around exactly one nested subtype describing what actually happened on the device. Across our captures we observed 25 different subtypes, and together they cover a lot of ground: process execution, device unlocks, ADB activity, app lifecycle events, boot integrity, Advanced Protection policy changes, Keystore activity, certificate installation, password changes, removable media, and Bluetooth connections.

The standout subtype is `app_process_start`, which records every process launched on the device along with its UID, PID, SELinux context, start time, and the SHA-256 hash of the binary that was executed. In practice, this moves investigators from asking what was installed to knowing what actually ran.

```
{
  "security_event": {
    "event_time": 1782839560465952259,
    "app_process_start": {
      "process": "com.whatsapp",
      "start_time": 1782839560465,
      "uid": 10381,
      "pid": 31469,
      "seinfo": "default:targetSdkVersion=36:complete",
      "sha256": "e76e163b709acc08eed2d4a6d7b3382254..."
    }
  }
}
```


ADB activity gets a family of subtypes of its own: `adb_shell_cmd`, `adb_shell_interactive`, `adb_sync_send_file`, and `adb_sync_recv_file`. One-shot shell commands are captured verbatim, file transfers are recorded with their send and receive paths, and in our data Intrusion Logging even captured the ADB pull of its own previous log bundle. That works in an investigator's favor when it comes to chain of custody, but it is worth remembering that anything you do on the device over ADB is written into the same record.


```
{
  "security_event": {
    "event_time": 1780955974504907973,
    "adb_shell_cmd": {
      "command": "rm /sdcard/Download/test_il.txt"
    }
  }
}
```


The one gap in the ADB picture is interactive shells: `adb_shell_interactive` records that a session was opened, but nothing typed inside it.


```
{
  "security_event": {
    "event_time": 1781041163047185656,
    "adb_shell_interactive": {}
  }
}
```

Boots are similarly recognizable. A reboot produced a tight burst of:

```
os_startup
user_restriction_added / user_restriction_removed
logging_started
crypto_self_test_completed
```

`os_startup` included Verified Boot state, such as `boot_state: "green"` and `verity_mode: "enforcing"`.

Some important details worth noting when parsing these logs.

1. Timestamp units differ.
    a. `dns_event` and `connect_event` use milliseconds.
    b. `adb_shell_cmd`, `adb_shell_interactive`, `security_event.event_time` uses nanoseconds.
    c. Some nested fields, such as `app_process_start.start_time`, go back to milliseconds.
2. Security events carry no explicit type field. The subtype is the nested key itself, the one that isn't `event_id` or `event_time`, so a parser has to derive it by elimination rather than read it from a field.
3. Some privacy protection is already applied before the analyst sees the data, for example Bluetooth MAC addresses were redacted within the log.

All of this adds up to a compact but rich audit trail. It falls well short of packet capture or full logcat, but deliberately so. Google has built a curated record of the activity Android itself considers most relevant to intrusion analysis, covering DNS lookups, network connections, process starts, unlocks, ADB activity, app changes, boot state, policy changes, and changes to what the device trusts. For defenders, it meaningfully expands what Android can prove after the fact.

What IL Captures, and What It Misses

What makes all this structure valuable is attribution. Android has always produced useful forensic traces, but they tend to be short-lived, fragmented, and hard to tie back to a specific app. Intrusion Logging changes that: a suspicious domain is no longer just a network indicator but one bound to the package that requested it, and a process is recorded with its hash, its launch time, and the SELinux context it ran in. In one session on our test device, IL captured a multi-minute ADB sequence used to investigate and remove an app, down to the individual package-manager commands and shell one-liners, the kind of reconstruction that was simply not possible before without live monitoring.

Taken together, these artifacts make IL strong for reconstructing a timeline of device access, app execution, network activity, ADB use, trust-store changes, credential changes, and boot integrity. It is not packet capture or full system logging, but it gives investigators a durable record of the events most likely to matter in a consensual spyware investigation.

IL does have limitations however.

The largest network blind spot we observed is encrypted DNS. IL records lookups that pass through Android's system resolver, so any app that does its own DNS-over-HTTPS or DNS-over-TLS resolution will not be recorded. We saw this clearly when testing with Chrome - IL captured the bootstrap lookup for the DoH infrastructure, and then went quiet while the individual queries travelled through the encrypted channel. It is worth noting, that when we enabled Android's system-wide Private DNS in strict mode, pointing the whole device at an encrypted DNS-over-TLS resolver, Intrusion Logging kept recording hostnames exactly as before. The system resolver still handles the lookup and logs it before encrypting the query to the upstream server, so turning on Private DNS for privacy will not impact any analysis.

A full-device VPN behaves the same way. With a commercial VPN connected and its own ad and tracker blocking left off, Intrusion Logging carried on recording hostnames with per-app attribution exactly as before, because the system resolver was still performing the lookups and simply routing them through the tunnel. What blinds Intrusion Logging is not encryption or tunnelling in itself, but an application resolving its own DNS in-process, whether that is a browser's built-in secure DNS or a VPN configured to run its own DNS filtering. Anything that leaves resolution to Android's system resolver leaves IL's view intact.

For plaintext system-resolver DNS the coverage is excellent, but analysts should treat the DNS layer as one-directional evidence, a hostname appearing in IL is solid proof of a lookup, while a hostname missing from IL does not mean it wasn't resolved. The VPN did change the connection stream, but in a way that helps rather than hinders. Each app's connections to its real destinations were still recorded and correctly attributed, the tunnel does not collapse them to a single address, in addition, the VPN client's own traffic to its gateway was plainly visible, a high volume of connections from the VPN app to a small set of external gateway addresses. So while Intrusion Logging cannot see inside the tunnel, it will readily show that a tunnel exists and which app is running it, which is itself a useful signal in an investigation.

The app lifecycle events have a gap of their own, installer source. A `package_installed` event records the package, version, and user, however it does not contain information about whether the app arrived from Google Play, a sideloaded APK, an enterprise channel, or ADB. In other words, IL will tell you that an app appeared on the device, but not where it was installed from.

Collection timing needs care too. Download & decrypt hands you the latest bundle that made it to cloud backup, not the live state of the phone, and across repeated collections we regularly found the newest events missing until the next backup window came around. For an incident responder collecting logs on first contact with a device, that can mean the freshest 12 to 24 hours simply are not there yet. It is worth documenting the collection time and the last event time as a matter of routine, and where circumstances allow, letting the device complete a backup before drawing any conclusions about recent activity.

Some event classes are simply absent. We ran positive-control tests where we toggled Wi-Fi and NFC inside the logging window, bracketed by ADB marker commands so we could be sure the window was live; the markers appeared in the logs, the Wi-Fi and NFC activity did not. So IL cannot currently tell an analyst which Wi-Fi networks a device joined, whether it passed through a rogue access point, or whether NFC was enabled during a suspicious window. Shutdowns are also missing, boots show up clearly, but nothing marks a power-off, which leaves analysts inferring shutdown times from the gaps.

Finally, expect noise. In one of our captures, loopback traffic dominated the `connect_event` stream, and on any primary device a large volume of benign app, system, and localhost activity is just the baseline. IL should be treated as an evidence source rather than an alert feed, the signal is there, but getting to it still takes filtering, enrichment, and correlation.

None of this diminishes what IL provides. It gives Android investigators a reliable backbone for the case timeline; the edges are where it needs help, with supporting telemetry to cover its blind spots and conventional forensic artifacts for the newest activity that has not yet reached the cloud backup.

Analysis Tools

As mobile threat hunting analysts, we recognise that the value of any new data source depends entirely on the tools available to parse it. Fortunately, the open-source forensic ecosystem is already evolving to meet this need, thanks to the collaborative efforts of Google and Amnesty International's Security Lab to successfully integrate support for Intrusion Logging into tools like AndroidQF[4] and the Mobile Verification Toolkit (MVT).[5]

Intrusion Logging support in MVT is very new, and while testing it we hit one bug worth flagging for anyone relying on the tool today. In the current release (2026.5.12) MVT preserves the raw `security_event` payloads, but its summary and timeline label every security event as `event_id` rather than its real subtype, so process launches, ADB commands, certificate installs, unlock attempts and boot events are all present but collapsed into a single bucket for triage.

We raised this with the project and found the fix had in fact already been merged upstream (PR #815),[6] so it will ship in the next release. If you are working with version 2026.5.12, classify security events from the raw payloads rather than the summary view until the next release lands.

Our own internal research and hands-on experience implementing Intrusion Logging on physical Pixel hardware have been instrumental in refining our Threat Hunter for Android desktop application, which now supports the collection and parsing of intrusion logs on Android.

Because an IL bundle is ultimately a set of newline-delimited JSON files, much of it is also amenable to manual analysis. However, as our findings detail, the inconsistent timestamp precision, non-chronological file naming and `event_id` quirks make purpose-built tooling strongly preferable to ad-hoc parsing.

4 AndroidQF

5 Mobile Verification Toolkit (MVT)

6 MVT PR PR #815

Conclusion

Intrusion Logging is genuine, meaningful progress in the fight against mobile spyware. For the first time, a user on a device which supports Advanced Protection's IL can opt in to a durable, tamper-resistant, consent-driven record of the events that matter most in an investigation or incident response.

Instead of relying only on volatile logs, app inventories, screenshots, or whatever artifacts happen to remain on disk, an investigator can ask more direct questions:

- Which app resolved a suspicious domain?
- Which processes actually received execution?
- Is the process recognizable across a larger data corpus for threat hunting?
- Was ADB used? If so, over what period of time?
- Was a new, unexpected certificate installed?
- Did the device boot in a healthy Verified Boot state?
- Were there unlock failures around the period of suspected access?

These questions focus an analysts' triage and enable the careful construction of a forensic timeline for any investigative efforts.

The limits are also important. IL does not capture everything. DNS-over-HTTPS can bypass its DNS view. It does not provide Wi-Fi history or installer source. Interactive ADB sessions are less visible than one-shot commands. The newest events may still be waiting for the next cloud backup when collection happens. Analysts should treat IL as a powerful new evidence source, not as a complete record of device activity.

That balance is what makes the feature credible. The value is not in claiming that Android can now detect every intrusion,but that Android can now preserve a much stronger factual record for users who choose to enable it before something happens.

The next step is operational maturity. Today, practical use still looks like a forensic workflow: user interaction on the device, then ADB or specialist tooling to retrieve and parse the bundle. That is workable for labs and incident responders, but it limits reach.

A user-consented, security-focused API for trusted defensive tools would make it easier for products like iVerify to collect and analyze IL data without weakening the privacy model that makes the feature defensible.

The payoff of that maturity is scale. IL's per-app DNS attribution is a natural join against threat-intelligence feeds, and the same applies to process hashes against known-good and known-bad. Making that routine would require several pieces: consent-driven collection across a managed fleet, an ETL stage that normalizes ILs' mixed timestamp units and nested security-event structure, and enrichment that reduces a large event stream to the items warranting analyst review.

Today enrolment is a manual, per-device action with no management hook we could find, which raises governance questions the ecosystem has not yet resolved. On enterprise-owned or device-owner-managed hardware, can enrolment be configured and enforced centrally, for instance through an MDM policy, so an organization can enable Intrusion Logging across a managed fleet and confirm it stays active? And on a bring-your-own-device population, where enrolment has to remain the user's own decision, how would meaningful coverage be achieved at scale without the ability to mandate it?

Google deserves credit for building this around consent and privacy from the start: encrypted storage, user-controlled release, long-term retention, and an event stream focused on security rather than device diagnostics and Android app development. Amnesty International's Security Lab also deserves recognition for helping push this kind of capability into a mainstream platform in a way that serves real investigative needs.

As rollout expands, Intrusion Logging should become one of the first artifacts collected in Android incident response. It will not be the whole story, but for the first time on Android, it may preserve enough of the story to know where to look next.