Swan

Swan Security Review: Entropy Generation in the Blockstream Jade

1. Summary

As part of standard response procedure to industry-wide incidents, Swan has undertaken a review of the hardware vendor related to its Swan Vault offering. Today, Swan Vault operates on Blockstream Jade, with a second vendor planned. More information on this setup is presented in our prior article on the matter.

This report presents an internal Swan security audit of random-number generation in the Blockstream Jade hardware wallet firmware following the Coldcard incident. Our report confirms findings by the Blockstream team, which indicate that "Jade does not have a downgraded random number generation path to fall back to, so this problem cannot occur. Jade fetches random numbers from the device itself, generated from internal hardware chip noise." The Jade generates randomness using CPU counters, battery state, ambient temperature, multiple images taken with the camera during boot, a built-in cryptographic-strength hardware number generator, and, optionally, entropy from the Blockstream companion app.

Our independent source review found no exploitable defect. We agree with Blockstream that Jade's entropy generation is sound as designed.

---

2. Scope and method

In scope

The audit covers all firmware code that generates, seeds, mixes, or consumes random values. This includes:

- All board variants: Jade v1.0, v1.1, v2, v2c (note, however, that we do not support v2c in Swan Vault).
- The vendored ESP-IDF bootloader components.
- The libjade host build and the QEMU build. Both are development-only.

Out of scope

- Hardware silicon internals beyond vendor documentation.
- The PIN-server backend service.
- The internals of libwally and libsecp256k1.

Method

The audit used multiple review passes, including both AI and human reviewers. When working with LLMs, it is important to note their non-determinism. To account for this, each pass used different prompts and AI models, and the final report consolidates dozens of individual passes by looking for convergence and agreement. Finally, four human reviewers with security expertise reviewed this report and the associated source code.

Reference commit

All line numbers refer to commit `cfce08ced9f58ed0244e516f1d7f0c61a82889ee`. Blockstream has since released a patch update with a number of improvements. We intentionally audited the pre-release Jade firmware to determine whether it had an entropy bug before any additional fixes.

---

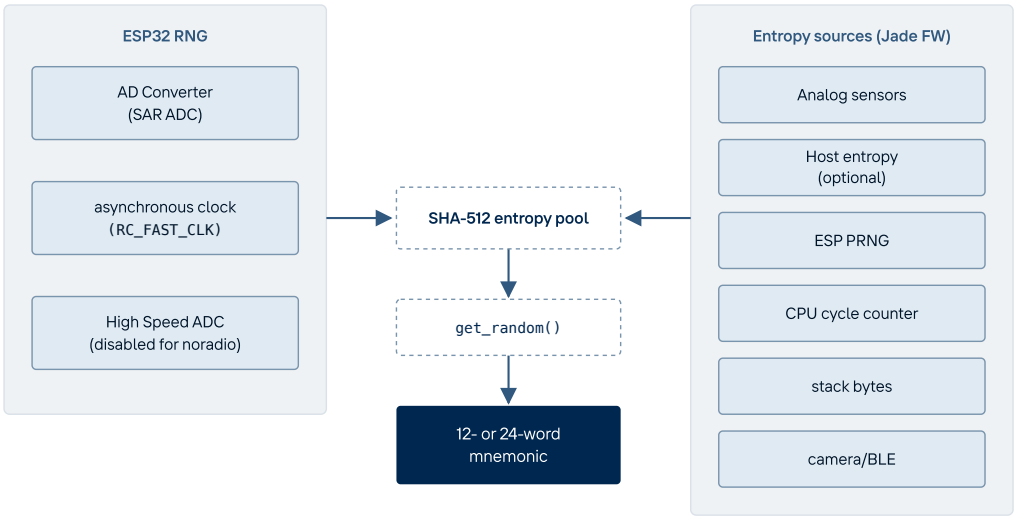
3. How Jade generates random numbers

Random numbers are used to protect the core properties of Bitcoin hardware wallets: the seed/recovery phrases, session keys, and encryption values. A weak random number can expose a private key.

The audit covers the Jade firmware at commit `cfce08c` (full hash `cfce08ced9f58ed0244e516f1d7f0c61a82889ee`). The build uses ESP-IDF v5.5.4, pinned in (`Dockerfile:9`).

Our findings indicate that every security-critical random value used in wallet operations comes from one source: the pool. Having all randomness come from a single pool is a good design that ensures that all paths follow the same secure protocol, unlike what was discovered in the Coldcard code, where some paths did not follow the same path.

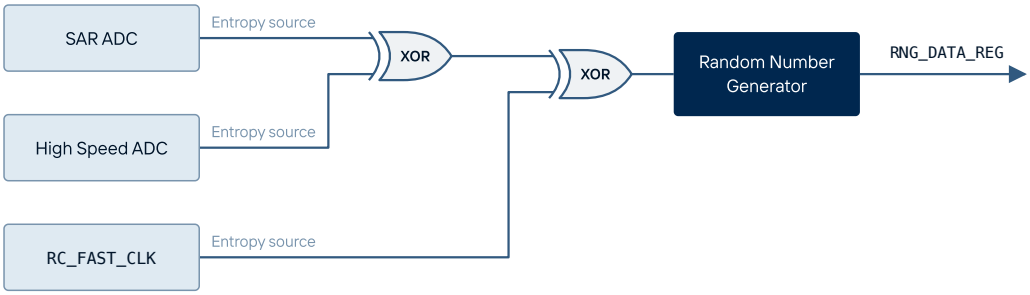
The pool is a SHA-512 accumulator inside `get_random_internal` (`random.c:71`), which operates as a cryptographic pseudo-random number generator (CSPRNG) with feedback. Firmware seeds the pool once at boot. The seed is a 32-byte value read from the hardware RNG while a true entropy source is active. Each later draw hashes the pool state and splits the digest: one half becomes the returned bytes, the other half becomes the next state.



Jade builds every secret from two parts: a hardware entropy source in the chip, and a software pool in the firmware. This section describes each part, then lists the properties the audit verified.

3.1 The hardware layer

The ESP32 Family chips contain a hardware RNG register called WDEV (also known as RNG_DATA_REG). Physical noise sources mix bits into this register continuously while a source is active. Three noise sources exist:



SOURCE	WHEN ACTIVE
The RF source (a high-speed ADC on the radio path)	Only while Wi-Fi or Bluetooth runs. Never active if “noradio” firmware is used.
The ADC source (a SAR analog-to-digital converter)	Only between `bootloader_random_enable()` and `bootloader_random_disable()`

SOURCE	WHEN ACTIVE
An asynchronous 8 MHz oscillator	Always on for ESP32-S3 and later chips. Only between bootloader_random_enable() and bootloader_random_disable() for ESP32 chips.

The high-speed ADC is coupled to the radio. The Technical Reference Manual states that it is enabled automatically when the Wi-Fi or Bluetooth module is enabled. On Jade, the Bluetooth controller starts in ble_init() during boot (main/main.c:268) only if the user enabled Bluetooth in a previous session (main/ble/ble.c:461). So, on a new or factory-reset unit, it does not start automatically. If the user enables Bluetooth in the settings menu, the controller starts at that point ble_start() (ble.c:476), and if the user disables Bluetooth, the controller stops, and the high-speed ADC stops with it. In --noradio builds, the controller never starts. This means the high-speed ADC is likely not used for entropy in some user scenarios during seed generation, but the other entropy sources are in use.

The firmware reads the register through `esp_random()`, which returns one 32-bit word. That function busy-waits before each read, so enough entropy enters the register first. The length of that wait is set per chip, and on plain ESP32, we find that it *may* be short, but this is unclear. Section 6 explains why the seed is still sound regardless. This wait is vendor-documented behavior in ESP-IDF v5.5.4.

3.2 The seed

The seed is the 32-byte boot value that starts the pool. Jade collects it early. `app_main` calls `random_start_collecting()` as its second statement (`main.c:301`), before any task exists.

That function runs three lines: `bootloader_random_enable()` (`random.c:275`), `esp_fill_random(entropy_state, sizeof(entropy_state))` (`random.c:276`), and `bootloader_random_disable()` (`random.c:277`). So the SAR ADC source is on for the whole read. The ESP-IDF second-stage bootloader also holds the SAR ADC source on for its entire run, from `bootloader_init.c:105` to `bootloader_utility.c:799`. The register is therefore well mixed before the app reads the seed.

The mutex that serializes the pool is created after the seed and before any task starts (`random.c:280`). Any draw that runs before seeding hits `JADE_ASSERT(rnd_mutex)` (`random.c:73`) and the device aborts.

After seeding, `random_full_initialization()` (`random.c:284`, called at `main.c:253`) runs `strengthen()`. This performs about one second of SHA-512 hashing over CPU cycle-counter jitter (`random.c:228`), then a sanity check (`random.c:180`). These functions also call into get_random_internal() several times, which also adds the sources described below.

3.3 The pool

The pool is the SHA-512 accumulator that produces all output. A draw is one call to `get_random_internal` (`random.c:71`). Each draw hashes, in order:

- Six power-sensor reads (`random.c:86–91`)
- The on-die temperature, on ESP32 only (`random.c:93`)
- The CPU cycle counter (`random.c:95`)
- Any caller-supplied data (`random.c:100`)
- The persistent 32-byte secret `entropy_state` (`random.c:105`)
- A counter that increments each call (`random.c:106`, `random.c:108`)
- A 64-byte stack buffer while it is still uninitialized (buf, declared at random.c:81, hashed at random.c:111)
- The same buffer a second time, after esp_fill_random(buf, sizeof(buf)) fills it with 64 bytes (random.c:115, hashed at random.c:116)

The 64-byte digest splits into two halves. Bytes 0-31 go to the caller (random.c:120-123). Bytes 32-63 become the new `entropy_state` (`random.c:127`). The scratch buffer is then wiped (`random.c:135`).

Requests larger than 32 bytes become successive 32-byte draws (`random.c:145`, `random.c:151`, `random.c:153`). `get_uniform_random_byte()` removes modulo bias with rejection sampling (`random.c:158–169`).

3.4 Entropy refeeds

refeed_entropy(data, len) (random.c:138) mixes caller-supplied data into the pool. This is an optional additional entropy source. Each refeed also re-samples the sensors and the hardware RNG/PRNG, independent of the payload size.

The firmware has six refeed call sites:

- strengthen(), once at boot inside random_full_initialization(). The payload is the 64-byte digest of the one-second SHA-512 stretch and its timing-jitter samples (random.c:260).
- random_sanity_check(), once at boot, directly after strengthen(). Adding a few bits of new entropy generated by two CPU cycle-counter reads (random.c:223).
- rnd_camera_feed(), at boot on units with a camera. The payload is one full raw image frame per call, for 11 calls (main.c:166).
- The dashboard screen, each time the GUI returns to it. The payload is a 4-byte FreeRTOS tick count (process/dashboard.c:2582).

- The add_entropy message, on receipt from a connected host. The payload is the host-supplied bytes (process/dashboard.c:401).
- The BLE stack, once every time there is a new advertising address (at least once at enable-time). The payload is the 6-byte BLE address (ble/ble.c:294).

3.5 Refeeds before wallet seed creation

On a new device, the pool receives these refeeds before the wallet seed is drawn (keychain.c:281):

- Always: the strengthen() refeed, the sanity-check refeed, and at least one dashboard tick refeed.
- On units with a camera: the 11 camera-frame refeeds.
- When a companion app is connected before setup: the add_entropy refeed.
- When the user enabled Bluetooth before setup: the BLE address refeed.

3.6 Verified security properties

The audit confirmed four properties of this design.

Forward secrecy. An attacker who dumps `entropy_state` from memory cannot reproduce earlier outputs. That would need a SHA-512 preimage.

Output and state separation. The bytes returned to the caller reveal nothing about the retained state, because the two halves come from one digest.

Additive-only mixing. External data enters the pool beside the secret and never replaces it. This covers host-supplied entropy from the `add_entropy` message (`dashboard.c:401`), camera frames (`main.c:166`), and Bluetooth addresses (`ble.c:294`). None of these inputs can force, bias, or read the pool. Empty payloads are rejected (`dashboard.c:394`).

No persistence. No RNG state survives a reboot. The pool seeds fresh every boot.

3.7 Host entropy

We investigated the Blockstream and Swan Vault apps to see whether they supplied the optional host entropy (add_entropy). Our findings indicated that Blockstream iOS and Android apps did supply this additional entropy, while the Blockstream Green desktop app and the Swan Vault web app did not. Both Blockstream and Swan have updated their desktop applications to add the additional entropy. Note that this entropy is defense in depth and is not required, given the sound entropy generation of the hardware device. Existing vaults do not need to do anything.

5. Entropy sources by hardware model

The firmware runs on several boards. Two separate properties change between them.

- Property A: does the board have an always-on hardware noise source? The ESP32-S3 does. The original ESP32 does not.
- Property B: do the per-draw sensor reads return real measurements? Jade v1.x boards return real values. Most S3 boards do not.

The hardware RNG is the WDEV register that `esp_fill_random` reads. On the ESP32-S3, an always-on oscillator feeds it. On the original ESP32, the register is only true-random when a radio runs. The ADC source is the SAR analog-to-digital converter, sampled at boot. The RF source is radio noise.

We initially identified that it may be possible that the app-side read of the entropy source (`random.c:275–277`) drains the hardware RNG register faster than the ADC source refills it due to timing of the hardware chips. However, on further examination, we came to the conclusion that this is a false positive. We reported the same to Blockstream, and they also arrived at a similar conclusion. This is due to the fact that the ADC source is enabled during the bootloader (enable `bootloader_init.c:105`; disable `bootloader_utility.c:799`). The register should therefore be well mixed before `app_main` starts. Furthermore, Jade's mixing in of many additional entropy sources more than covers this issue.

Source availability

SOURCE	JADE V1.0 / V1.1 (ESP32)	JADE V2 (ESP32-S3)	JADE V2C (ESP32-S3)
Boot seed via ADC source	Yes	Yes	Yes
Per-draw hardware RNG, radios idle	Pseudo-random	True-random	True-random
Per-draw hardware RNG, Bluetooth active	True-random	True-random	True-random
Camera-frame mixing at boot	Yes	Yes	No camera
Bluetooth-address mixing	Yes	No	No

Bluetooth-address mixing is ESP32-only. The call at (`ble.c:294`) sits inside an ESP32-only compile guard.

Per-draw sensor reads

Each draw reads six sensors and mixes them into the pool (`random.c:86–91`).

BOARD	REAL SENSOR READS
Jade v1.0 / v1.1 (`jadev10.inc`, `jadev11.inc`)	All six sensors return real ADC measurements
Jade v2 (`jadev20.inc`)	Battery voltage only, and only with a battery present; the other five return constant 0 (`jadev20.inc:265–273`)
Jade v2c	None; all six return 0. Kconfig defines v2c with no camera and no battery
Community boards (`minimal`, `m5stackcores3`, `twatchs3`)	None; all six return 0

The pool's security never depends on the sensors. They are defense in depth. On every model, the hardware RNG is the entropy source. The models, therefore, differ only in which defense-in-depth inputs they carry, and no model is weaker for the difference.

The code comment at (`random.c:85`) says the sensor data comes from an `axp192` power-management chip. On the v2 family, the companion chip is an STM32, and most reads return 0.

Source references

This audit reviewed the Jade firmware at commit `cfce08ced9f58ed0244e516f1d7f0c61a82889ee` of `blockstream/jade`. ESP-IDF code is cited at tag v5.5.4, the version the firmware pins. This includes components/esp_hw_support/hw_random.c, which holds the per-chip esp_random() wait constants, and the ESP-IDF API reference page quoted in section 6: https://docs.espressif.com/projects/esp-idf/en/v5.5.4/esp32/api-reference/system/random.html

Each cited file was fetched from `raw.githubusercontent.com` at that commit. Every fetched copy was confirmed byte-identical to the reviewed tree by SHA-256 comparison. Each cited line was then re-read from the fetched copy. All quoted code and comments in this report are verbatim.

FILE	ROLE
`main/random.c`	entropy pool, seed, uniform-byte helper
`main/keychain.c`	recovery phrase and EC key generation; stored-wallet encryption

FILE	ROLE
`main/main.c`	boot sequence; camera mixing
`main/process/pinclient.c`	PIN-server handshake draws
`main/process/get_bip85_entropy.c`	BIP85 export wrapper
`main/process/dashboard.c`	host `add_entropy`; per-message blinding
`main/process/mnemonic.c`	recovery-phrase flow and CI guard
`main/attestation/attestation.c`	attestation key and IV
`main/rsa.c`	RSA-PSS salt; deterministic BIP85 keygen
`main/wallet.c`	deterministic BIP85 derivation
`main/utils/wally_ext.c`	libsecp256k1 blinding
`main/aes.c`	AES-CBC IV
`main/identity.c`	RFC6979 signing; EC blinding
`main/gui.c`	decoy-glyph rendering
`main/ble/ble.c`	Bluetooth address mixing; default-off gate
`main/power/jadev10.inc`, `jadev11.inc`, `jadev20.inc`	per-board sensors
`main/Kconfig.projbuild`	board capabilities; amalgamated-build default
`libjade/libjade.c`	development host build
`bootloader_components/.../bootloader_random.c`	ADC source; bootloader reader
`bootloader_components/.../bootloader_init.c`	bootloader enables the source
`bootloader_components/.../bootloader_utility.c`	bootloader disables the source
`Dockerfile`	ESP-IDF version pin