

SECURITY AUDIT REPORT

MagicBlock vrf-program Solana program

by  **ARDA**

on September 6, 2026



Table of Contents

Disclaimer	3
Terminology	4
Objective	4
Summary	5
Inherent Risks	6
Code Issues & Recommendations	9
C1: Attacker can spam a queue and brick it forever	9
C2: Queue can be too big and all operations exceed Solana CU limit	11
C3: Requests for randomness can continuously be added and oracle can't ever close its queue	12
C4: Oracle queue can be delegated to Ephemeral Rollup while already used on mainnet	13
C5: Oracle queue can be undelegated from Ephemeral Rollup	14
C6: No dissuasive mechanism against oracle withholding VRF fulfillment	15
C7: Request expiry assumes fixed slot duration but this will change in the future	17
C8: Misleading and mishandled guard "next <= cursor" in queue iterations	20
C9: No explicit type validation on VRF proof elements	21
C10: Cooldown on queue registration adds complexity without benefit	23
C11: Program detection via the executable flag is not fully reliable	24
C12: Reading a request's callback data returns empty instead of an error when the range is invalid	26
C13: Queue scanning logic is duplicated across several methods	28
C14: Unnecessary mixing of oracle insertion and removal logic	32
C15: Redundant and complex request insertion logic	33
C16: Unnecessary use of unsafe Rust	34
C17: Misleading terminology and code formatting	36
C18: Unnecessary arguments	38
C19: Redundant stripping of the queue discriminator	39
C20: Errors in comments	40
C21: Dead and unused code	42

Disclaimer

The report makes no statements or warranties, either expressed or implied, regarding the security of the code, the information herein or its usage. It also cannot be considered as a sufficient assessment regarding the utility, safety and bugfree status of the code, or any other statements.

This report does not constitute legal or investment advice. It is for informational purposes only and is provided on an "as-is" basis. You acknowledge that any use of this report and the information contained herein is at your own risk. The authors of this report shall not be liable to you or any third parties for any acts or omissions undertaken by you or any third parties based on the information contained herein.

Terminology

Code: The code with which users interact.

Inherent risk: A risk for users that comes from a behavior inherent to the code's design.

Inherent risks only represent the risks inherent to the code's design, which are a subset of all the possible risks. **No inherent risk doesn't mean no risk.** It only means that no risk inherent to the code's design has been identified. Other kind of risks could still be present. For example, the issues not fixed incur risks for the users, or the upgradability of the code might also incur risks for the users.

Issue: A behavior unexpected by the users or by the project, or a practice that increases the chances of unexpected behaviors to appear.

Critical issue: An issue intolerable for the users or the project, that must be addressed.

Major issue: An issue undesirable for the users or the project, that we strongly recommend to address.

Medium issue: An issue uncomfortable for the users or the project, that we recommend to address.

Minor issue: An issue imperceptible for the users or the project, that we advise to address for the overall project security.

Objective

Our objective is to share everything we have found that would help assessing and improving the safety of the code:

1. The **inherent risks** of the code, labelled R1, R2, etc.
2. The **issues** in the **code**, labelled C1, C2, etc.
3. The **issues** in the **testing** of the code, labelled T1, T2, etc.
4. The **issues** in the **other** parts related to the code, labelled O1, O2, etc.
5. The **recommendations** to address each issue.

Summary

Blockchain: Solana

Initial scope

- **Repository:** <https://github.com/magicblock-labs/solana-vrf/>
- **Commit:** 8397569bd47388f400e7f2f32360fbb5dcac165b
- **Path(s):**
./program/src

Final scope

- **Repository:** <https://github.com/magicblock-labs/solana-vrf/>
- **Commit:** fecd04ca27213069e3ebb4a2435acbddf6fa193e
- **Path(s):**
./program/src

5 inherent risks in the final scope

2 issues in the final scope

21 issues reported in the initial scope and 2 remaining in the final scope.

Severity	Reported			Remaining		
	Code	Test	Other	Code	Test	Other
Critical	0	0	0	0	0	0
Major	1	0	0	0	0	0
Medium	6	0	0	0	0	0
Minor	14	0	0	2	0	0

Inherent Risks

R1: Users have no guarantee that a randomness request will be successfully answered.

This is because the randomness for a request is produced and delivered by an oracle, and nothing forces the oracle to respond. A request might go unanswered or fail to be answered for several reasons, for example:

- The oracle is offline.
- The oracle chooses not to deliver a result it does not like.
- The oracle does not pay enough gas for the delivery transaction, so it fails.
- The callback instruction contains the oracle address as an `AccountMeta`, which is forbidden by the VRF program.

However, in practice, several protections are implemented to reduce the risk of randomness withholding:

- (restricted set) Oracles are not permissionless: only the program's upgrade authority can add or remove one, so requests are served by a curated set of operators rather than by anyone.
- (reputation cost) Oracle behavior is visible on-chain and per-queue, so a misbehaving oracle could lose integrators who switch queues.
- (financial cost) If a request is not fulfilled within the fulfillment period, the oracle can be slashed: the request fee is burned and the oracle's stake is slashed after the deadline.

R2: Users might be unable to submit a randomness request when the oracle's queue is full.

This is because each oracle keeps pending requests in a queue of limited size. If that queue is already full when a user submits a request — because many

requests are waiting, or expired ones have not yet been cleared — the new request is rejected and the transaction fails.

Note that the risk is partially mitigated by the following:

- An oracle can deploy as many queues as it wants if it observes that the current ones are full.
- Requests expire after 120 seconds and purging them is permissionless.

R3: Projects relying on the randomness should be aware that the random value is briefly visible to everyone before its result is provided to the project's program, and so others can act on it in advance.

This is because a request and the delivery of its result happen in two separate steps, and once the delivery transaction is signed by the oracle, it sits in the validators' transaction pool until it is executed.

Until then, anyone can see the outcome of the randomness and decide to perform actions based on it before it lands in the destination program — for example, moving their funds ahead of an outcome they expect to be unfavorable.

R4: Projects should be aware that the oracle knows requests' results before anyone else, and could thus act on outcomes in advance.

This is because the oracle produces the randomness itself, so for any request it can work out the result before revealing it. A dishonest oracle can use this in several ways:

- It can act on a result before anyone else — for example taking a position, in this or another project, based on an outcome that it alone knows so far.
- It can choose which requests to answer based on the result it computes, serving the ones that suit it and leaving the others unanswered.

- If a project lets the requester choose or influence the input attached to a request (the `caller_seed` argument), then the oracle (acting as the requester, or working with it) can try inputs until it produces a request with a desired outcome.

In practice, the risk is partially mitigated by the same factors as in R1: in particular, oracles are whitelisted rather than permissionless, and slashing deters them from withholding fulfillment.

R5: Users paying a higher fee for priority requests have no guarantee they will be executed faster.

This is because the order in which requests are served is completely trusted: it is decided by the oracle off-chain, and nothing is enforced on-chain. Paying the higher priority fee only increases the amount the requester pays; it does not oblige the oracle to serve that request before others.

Code Issues & Recommendations

C1: Attacker can spam a queue and brick it forever

Severity: Major

Status: Fixed

Location

```
program/src/purge_expired_requests.rs  
    process_purge_expired_requests
```

Description

Current behavior: `process_purge_expired_requests` removes expired items one at a time with a $O(n^2)$ complexity:

- It iterates over all items in the queue ([link](#)).
- For each item, it calls `get_item_by_index(i)`, which also iterates over the queue. Moreover, for each expired item, `remove_item(i)` is called and iterates over the queue from the start and then calls `trim_trailing_holes()` ([link](#)), which itself iterates over the entire queue again.

By inserting as many fake items as possible in a queue, an attacker could make the purge irreversibly fail by exceeding Solana's 1.4M CU limit per transaction.

On mainnet, the queue has size 30,000 bytes ([link](#)), so 29,976 bytes are available for requests (after stripping away the discriminator and Queue header). To maximize the number of inserted requests, the attacker makes requests with no discriminator, no arguments and no callback accounts. So each request reduces to 96 bytes for filling the `QueueItem` struct.

In total the attacker can fit $29976 / 96 = 312$ requests in the queue. This is enough to brick the queue forever because:

- On Solana mainnet, reading or writing an item conservatively costs 20 CU.
- If we only add the costs of `trim_trailing_holes`: it is called once per item in the queue, i.e. 312 times, and each time it reads all the items in the

queue. This in itself costs $312 * 312 * 20 = 1946880$, exceeding the 1.4M CU limit.

- Even without `trim_trailing_holes`, for each of the 312 items we call `get_item_by_index(i)` and `remove_item(i)`. Each method iterates until it finds the item. Therefore, together these methods have a conservative cost of $2 * \sum_{i=0}^{312} (312-i) * 20 > 1.95M$ CU, exceeding the 1.4M CU limit.

For each request, the attacker pays the oracle fee of 500000 lamports, and at most the same fee for the transaction. So 1M lamports per request, which is ~0.3 SOL in total.

Expected behavior: It should always be possible to remove expired items from the queue, no matter the number of items in the queue.

Worst consequence: An attacker fills the queue with a maximal number of non-executable items. From there, the queue is stuck and new requests from projects for randomness systematically fail. This might have unwanted impact on their internal logic and they also might turn to competitors.

Recommendation

We suggest introducing a new instruction

`process_purge_specific_request(index)` which takes as argument a precise logical index in the queue and only removes this item (if expired) without trimming holes.

Moreover, we would apply the recommendation to C2: Queue can be too big and all operations exceed Solana CU limit, i.e. impose an upper-bound on the queue size, ensuring that normal $O(n)$ iterations never exceed the 1.4M CU limit.

Optionally, here are possible reductions of `process_purge_specific_request` computational cost:

- Only call `trim_trailing_holes` once at the end, rather than on every removal.
- Do not separate `get_item_by_index` and `remove_item`. Instead, we can just call a single function `remove_item_by_index_if_expired` which iterates until it finds the item and removes it if it has expired.

C2: Queue can be too big and all operations exceed Solana CU limit

Severity: Medium

Status: Fixed

Location

`api/src/state/queue.rs`

Description

The oracle queue's variable region has no enforced upper bound on how many items it can hold, while every queue operation iterates over all items:

`scan_for_reusable_span` ([link](#)), `trim_trailing_holes` ([link](#)),
`get_item_by_index` ([link](#)), `remove_item` ([link](#)), and `find_item_by_id` ([link](#)).

Therefore, if the queue is created with too big a size ([link](#)), then some operations could fail by exceeding Solana's `1.4M` compute-unit (CU) limit.

Recommendation

In `process_initialize_oracle_queue`, we suggest enforcing a conservative maximum queue size `MAX_QUEUE_SIZE` expressed in bytes. For example, given a conservative estimate of `500` CU per item treatment, we would cap the queue to $1.4M / 500 = 2800$ requests, which would correspond to

`MAX_QUEUE_SIZE = 2800 * 96 = 268800`.

C3: Requests for randomness can continuously be added and oracle can't ever close its queue

Severity: Medium

Status: Fixed

Location

program/src/close_oracle_queue.rs
process_close_oracle_queue

Description

Current behavior: `process_close_oracle_queue` refuses to close a queue if it holds pending requests:

```
if !queue_acc.is_empty() {  
    return Err(EphemeralVrfError::QueueNotEmpty.into());  
}
```

This is to ensure that all pending requests, which were already paid by the projects, are fulfilled before the queue is closed.

However, this means that anyone can perpetually add requests to prevent the queue from being closed.

Expected behavior: The oracle should be able to stop accepting new requests, drain the queue, and close it.

Recommendation

We suggest adding an oracle-gated pause mechanism to the queue, blocking further requests.

For example, we can store a `paused` field in the queue header's reserved padding (`_padding`) to preserve backward compatibility, set and cleared by dedicated instructions `pause` and `unpause` that require the oracle's signature, and have `process_request_randomness` reject inserts into a paused queue.

C4: Oracle queue can be delegated to Ephemeral Rollup while already used on mainnet

Severity: Medium

Status: Fixed

Location

```
program/src/delegate_oracle_queue.rs  
    process_delegate_oracle_queue
```

Description

Current behavior: `process_delegate_oracle_queue` delegates the Oracle queue account to the Ephemeral Rollup (ER) without checking whether the queue already holds active items. However, this means that the queue might already be integrated with projects on Solana mainnet, and the delegation to ER would now make all these integrations fail.

Expected behavior: Delegation is a one-time setup step that should only be permitted when the queue is initialized.

Recommendation

We suggest restricting delegation to queues that have never been used, which can be done by checking (i) that all bytes of the queue variable region (following the header) are zeros, and (ii) `item_count == 0` (this is mostly defensive).

Note that it would not be sufficient to only verify that the queue *currently* has no items (e.g. only checking `item_count == 0`), because it could still be the case that some projects are actively relying on this queue on Solana mainnet.

C5: Oracle queue can be undelegated from Ephemeral Rollup

Severity: Medium

Status: Fixed

Description

Current behavior: The program allows undelegating an oracle queue that was delegated to the Ephemeral Rollup (ER). However, undelegation could pose a risk for existing integrations on the ER.

Expected behavior: Queue delegation is meant to be one-way: once a queue is delegated to the ER, it should be integrated with projects on the ER only, and never return to Solana mainnet.

Recommendation

We suggest removing the undelegation path entirely — i.e. both instructions `process_undelegate_oracle_queue` and `process_undelegation` .

C6: No dissuasive mechanism against oracle withholding VRF fulfillment

Severity: Medium

Status: Fixed

Description

Current behavior: The oracle can decide to not execute a specific randomness request, and will incur no cost from doing so.

Indeed, fulfillment is purely voluntary:

- Either the oracle chooses to fulfill the request by calling `process_provide_randomness`,
- Or it chooses to not fulfill the request, which then sits until it is purged and the requester's paid fee goes to the oracle even though the request was never served ([link](#)).

Nothing on-chain compels any oracle to respond or bounds the delay; the `priority_request` flag is only a higher fee tier, not an ordering or delivery guarantee.

Expected behavior: The project's team wants to investigate on-chain dissuasive mechanisms against an oracle withholding the fulfillment of requests. For example, an oracle could be economically bound to fulfill the request within a bounded time, and a requester would have on-chain recourse (a refund or an oracle penalty) when fulfillment does not occur.

Worst consequence: An oracle regularly avoids fulfilling requests from a project, whose functioning is degraded.

Recommendation

We suggest investigating a slashing mechanism for an oracle not fulfilling requests. If the project is interested in implementing such an approach, we suggest reaching out to the auditor to discuss it in more detail and to review the slashing logic.

At a high level, the slashing mechanism could work in the following way:

- A separate staking program can be introduced, enforcing staking requirements and slashing amounts. Oracles must register and stake there before they can be whitelisted in the VRF.
- A fallback *deposit* mode can be introduced, where the oracle merely deposits the randomness in the VRF program, and does not execute the callback CPI. Once deposited, anyone can call the execution path.
- The deposit mode is *opt-in*, i.e. a project allows it explicitly when performing a randomness request, e.g. by setting a boolean `deposit_randomness_allowed` as part of the request.
- If a request for which `deposit_randomness_allowed == true` does not have randomness deposited within a fixed delay (which should be shorter than the expiration period), then the oracle can be slashed.

Note: The reason for introducing the deposit mode is to untie the fulfillment of a VRF request from the callback CPI execution. Indeed, the callback CPI can fail for an arbitrary reason, therefore in the current mode it is impossible to determine whether a request is not fulfilled because the oracle withholds the fulfillment or because the CPI callback fails.

Resolution notes

The team implemented a novel slashing mechanism, which was reviewed by us, but not audited in full depth.

C7: Request expiry assumes fixed slot duration but this will change in the future

Severity: Medium

Status: Fixed

Location

`api/src/consts.rs`

Description

Request expiry is measured as a fixed number of slots, given by the constant `QUEUE_TTL_SLOTS = 240` ([link](#)), with a comment stating the following intent: "about 2 minutes on Solana", assuming approximately 500ms per slot.

However, slot duration might not stay constant over time; in fact, it is planned to be reduced over future Solana upgrades ([link](#)), and whenever it will occur, requests would expire too quickly.

Recommendation

In `process_purge_expired_requests`, we suggest assessing the age of a request using durations in timestamps (instead of slots). For this, we can convert elapsed slots into elapsed seconds as follows:

```
// The in-seconds replacement of QUEUE_TTL_SLOTS = 240;
pub const QUEUE_TTL_SECONDS: 64 = 120

pub fn process_purge_expired_requests(...) {

    let clock = Clock::get()?;
    let schedule = EpochSchedule::get()?;
    let epoch_start_slot =
        schedule.get_first_slot_in_epoch(clock.epoch);
    let epoch_start_ts = clock.epoch_start_timestamp
    let elapsed_slots = (clock.slot - epoch_start_slot).max(1);
    let elapsed_secs = clock.unix_timestamp - epoch_start_ts;

    while i < queue_acc.len() {
        let item = queue_acc.get_item_by_index(i)?;
        let age_secs =
            (clock.slot - item.slot) * elapsed_secs / elapsed_slots;
        if age_secs > QUEUE_TTL_SECONDS { /* expired */ }
        ...
    }
    ...
}
```

Note: This approach was designed with backward compatibility in mind, and therefore works well even if queues contain items when the upgrade occurs, and adjusts to future changes of slot duration. There is however the caveat that rounding errors are significant when a new epoch starts, leading to overestimations of the requests' time-to-live (TTL):

- In the interval `[0s, 1s)`, `elapsed_secs` is rounded to `0` (and so `age_secs = 0`), leading to a worst-case `-100%` underestimation of the actual age of requests, and no requests can be closed.
- In the interval `[1s, 2s)`, `elapsed_secs` is rounded to `1`, leading to a worst-case `-50%` underestimation of the actual age of requests, and requests' TTL is overestimated by up to `100%` (`240` seconds instead of `120` seconds).
- In the interval `[2s, 3s)`, `elapsed_secs` is rounded to `2`, leading to a worst-case `-33%` underestimation of the actual age of requests, and requests' TTL is overestimated by up to `50%` (`180` seconds instead of `120` seconds).
- In the interval `[19s, 20s)`, `elapsed_secs` is rounded to `19`, leading to a worst-case `-5%` underestimation of the actual age of requests, and

requests' TTL is overestimated by up to 5% (126 seconds instead of 120 seconds).

- As time passes, the rounding error quickly becomes negligible. Already 1 minute after the epoch starts, the worst-case TTL overestimation is 122 seconds instead of 120 seconds.

C8: Misleading and mishandled guard "next <= cursor" in queue iterations

Severity: Minor

Status: Fixed

Location

api/src/state/queue.rs

Description

All queue scanning loops (`scan_for_reusable_span` , `trim_trailing_holes` , `iter_items` , `get_item_by_index` , `remove_item` , and `find_item_by_id`) advance through items using `item_next` , which returns (`link`):

```
next = align_up(cursor + size_of::() + ..., align)
```

Since `size_of::()` is `96` , we see that the returned `next` value is always strictly greater than `cursor` .

However, each scanning loop nonetheless guards against `next <= cursor` and, when that happens, silently breaks out of the loop instead of reporting an error.

Not only is this `next <= cursor` requirement misleading (as it can never fire), but more importantly, silently breaking the loop could be harmful if it ever fires. For example, in `trim_trailing_holes` the early break computes a `last_used_end_aligned` that omits any live items past the break point and then shrinks `header.cursor` down to it (`link`), truncating the logical queue.

Recommendation

We recommend removing the `next <= cursor` guards from all functions.

Alternatively, if the project prefers to keep the guards, we suggest making the instruction fail instead of silently breaking the loop.

C9: No explicit type validation on VRF proof elements

Severity: Minor

Status: Won't fix

Location

```
api/src/verify.rs  
    verify_vrf
```

Description

Current behavior: The VRF proof is given by the oracle as the following raw elements: the two commitment points `commitment_base_compressed` and `commitment_hash_compressed` and the scalar `s` ([link](#)). At this point, there is no type guarantee that the two commitments are valid Ristretto points, and that `s` is a canonical scalar below the group order.

However, these type guarantees are never explicitly enforced by any check. Rather, the method `verify_vrf` directly uses elements in curve operations without validating them first.

Fortunately, checks are made internally in algebraic operations `multiply_ristretto` and `add_ristretto`, which reject malformed inputs.

However, the lack of explicit checks is error-prone and could lead to missing validation in future code changes, e.g. if the algebraic operations ever remove the internal checks, or if an element is used in the VRF program but not in any algebraic operation.

Expected behavior: The program should explicitly reject any non-canonical Ristretto point and any non-canonical scalar before using them in the curve operations, so that input validation does not depend on the internal behavior of `multiply_ristretto` and `add_ristretto`.

This would match with the specifications of RFC 9381 ([link](#)) which prescribe explicit checks when converting raw proof elements into scalars and curve points.

Recommendation

We suggest explicitly validating the proof inputs at the start of `verify_vrf` :

- Confirm both `commitment_base_compressed` and `commitment_hash_compressed` are valid Ristretto points, for example by calling `validate_ristretto` ;
- Confirm `s` is a canonical scalar, for example, by returning `false` when:

```
Scalar::from_canonical_bytes(s.0).into_option().is_none()
```

Resolution notes

The team acknowledged the issue and decided to keep the code as is because:

- All VRF elements are already implicitly validated in internal helpers.
- Adding explicit validations would cost more compute units.

C10: Cooldown on queue registration adds complexity without benefit

Severity: Minor

Status: Fixed

Location

```
program/src/initialize_oracle_queue.rs  
    process_initialize_oracle_queue
```

Description

Once an oracle is whitelisted by the VRF program admin, there is a cooldown of 200 slots (about 1m40s) before the oracle can register a queue ([link](#)).

This cooldown was added in case the oracle whitelisting would become permissionless in the future, to mitigate the case where a malicious oracle would quickly whitelist himself and register a queue to fulfill a specific request for randomness. However, in the current model, it serves no purpose.

Moreover, even in the hypothetical situation where oracle whitelisting becomes permissionless, this cooldown might not be useful or efficient:

- The calling program specifies the queue for its randomness, therefore if the malicious oracle does not control the calling program, he has no way to manipulate the randomness.
- In case the malicious oracle also controls the queue asked by the calling program, then the cooldown is not really a protection anyway because the malicious oracle could always pre-register queues 200 slots upfront.

Recommendation

We suggest removing the 200-slot registration cooldown from

`process_initialize_oracle_queue`, as well as related code, e.g. the following doc-comment:

```
/// - The Oracle must have been registered for at least 200 slots.
```

C11: Program detection via the executable flag is not fully reliable

Severity: Minor

Status: Fixed

Location

```
program/src/provide_randomness.rs  
    has_disallowed_executable_callback_account
```

Description

Current behavior: `has_disallowed_executable_callback_account` classifies a callback account as a program by reading its `executable` flag ([link](#)), and this helper is used as a defensive filter in the legacy global-identity branch to reject disallowed executable accounts in the callback ([link](#)).

However, the `executable` flag is not a reliable signal for whether an account is a program:

- Runtime checks on this flag are in fact completely removed ([link](#)).
- Legacy v1/v2 programs that were not deployed up to their pre-defined target size have `executable == false` while being executable.
- Some future BPF loaders might not even set this flag.

Although there are no exploitable problems in the specific current context (code and allowed programs `ALLOWED_EXECUTABLE_CALLBACK_ACCOUNTS`), this is a protection which is error-prone and non-robust to future changes.

Recommendation

We suggest making `has_disallowed_executable_callback_account` more robust, by complementing the `executable` flag check with a check that the account is owned by a BPF loader:

```

const LOADERS: = [
    // List of pubkeys of BPF loaders
    pubkey!("NativeLoader11111111111111111111111111111111"), // native
    pubkey!("BPFLoader11111111111111111111111111111111"), // v1
    pubkey!("BPFLoader21111111111111111111111111111111"), // v2
    pubkey!("BPFLoaderUpgradeable11111111111111111111111111111111"), // v3
    pubkey!("LoaderV4111111111111111111111111111111111111111111111111111111111111111"), // v4
]

fn is_program(a: &AccountInfo) -> bool {
    a.executable || LOADERS.contains(a.owner)
}

```

Note that, strictly speaking, `is_program` captures a superset of executable programs, as it would also return `true` for program data accounts and buffer accounts, because they are owned by BPF loaders. This is acceptable because there is no reason for any legacy integration to involve such program data accounts and buffer accounts in VRF callbacks.

C12: Reading a request's callback data returns empty instead of an error when the range is invalid

Severity: Minor

Status: Fixed

Location

`api/src/state/queue.rs`

Description

Current behavior: Each queued request stores its callback data (a discriminator, its account metas, and its arguments).

The 3 functions that read this data (`callback_discriminator` , `account_metas` and `callback_args`) return an empty slice when the requested range would fall outside the queue account (`end > acc.len()`), instead of returning an error ([link](#)). However, returning empty data hides the problem rather than stopping it, and might lead to an unwanted callback execution.

Furthermore, `account_metas` and `callback_args` also return empty data when the starting pointer of the data exceeds its end (`start > end`), which again hides a problem rather than stopping it.

Finally, the two guards above are defensive-only, as the situations they protect against can't ever occur:

- `end > acc.len()` can't occur, because an item can only be inserted in the queue through `write_item_at` , where there is an explicit protection against `end > acc.len()` .
- `start > end` can't occur, because `end` is defined in the same function as `start` plus an unsigned integer.

Recommendation

In `callback_discriminator` , `account_metas` and `callback_args` , we suggest reverting whenever either `end > acc.len()` or `start > end` occurs.

Alternatively, we can simply delete those guards, as there are explicit guarantees that they can never occur.

C13: Queue scanning logic is duplicated across several methods

Severity: **Minor**

Status: **Fixed**

Location

api/src/state/queue.rs

Description

In *queue.rs*, 6 methods (`scan_for_reusable_span` , `trim_trailing_holes` , `iter_items` , `get_item_by_index` , `remove_item` , and `find_item_by_id`) re-implement the same queue traversal logic. Essentially every copy has the following pattern:

```
let mut cursor = Self::items_start();
let end = core::cmp::min(self.acc.len(), self.header.cursor as
usize);
let align = core::mem::align_of::<QueueItem>();
let item_size = size_of::<QueueItem>();

while cursor + item_size <= end {
    let bytes = &self.acc[cursor..cursor + item_size];
    let item = Self::read_item_unaligned(bytes);
    let next = Self::item_next(cursor, &item, align);

    if next <= cursor {
        break;
    }

    // Function-specific logic here:
    // if condition(item) {
    //     do something
    // }

    cursor = next;
}
```

None

This traversal is delicate logic: it mixes raw byte offsets, unaligned reads, and alignment rounding, therefore copying it six times increases the risk surface.

Any later change to the traversal has to be re-applied to all six copies; missing one could break a queue invariant.

Recommendation

We suggest introducing a common helper `scan_items` that can be used in all 6 functions, taking care of the iteration logic, keeping only the specific "search" and "modification" logic separate. Here is a sketch of how this could look:

```
fn scan_items<R>(<
    &self,
    mut visit: impl FnMut(usize, usize, usize, &QueueItem) ->
Option<R>,
) -> Option<R> {
    let mut used_index = 0usize;

    let mut cursor = Self::items_start();
    let end = core::cmp::min(self.acc.len(), self.header.cursor as
usize);
    let align = core::mem::align_of::<QueueItem>();
    let item_size = size_of::<QueueItem>();

    while cursor + item_size <= end {
        let bytes = &self.acc[cursor..cursor + item_size];
        let item = Self::read_item_unaligned(bytes);
        let next = Self::item_next(cursor, &item, align);

        if let Some(result) = visit(used_index, cursor, next, &item) {
            return Some(result);
        }
        if item.used == 1 {
            used_index += 1;
        }

        cursor = next;
    }

    None
}
```

```

fn scan_for_reusable_span(&self, required_span: usize) -> QueueScan
{
    let mut last_used_end_aligned = Self::items_start();
    let mut reusable_span = None;

    self.scan_items(|used_index, item_pos, next, item| {
        if item.used == 1 {
            last_used_end_aligned = next;
        } else if reusable_span.is_none() && next - item_pos ==
required_span {
            reusable_span = Some(ReusableSpan {
                item_pos,
                logical_index: used_index,
            });
        }
        None::<()>
    });

    QueueScan {
        last_used_end_aligned,
        reusable_span,
    }
}

```

```

fn trim_trailing_holes(&mut self) {
    let mut last_used_end_aligned = Self::items_start();
    self.scan_items(|_, _, next, item| {
        if item.used == 1 {
            last_used_end_aligned = next;
        }
        None::<()>
    });

    if (last_used_end_aligned as u32) < self.header.cursor {
        self.header.cursor = last_used_end_aligned as u32;
    }
}

```

```
pub fn iter_items(&self) -> impl Iterator<Item = QueueItem> + '_ {
    let mut out = Vec::new();
    self.scan_items(|_, _, _, item| {
        if item.used == 1 { out.push(*item); }
        None::<()>
    });
    out.into_iter()
}
```

```
pub fn get_item_by_index(&self, index: usize) -> Option<QueueItem> {
    self.scan_items(|current, _, _, item|
        (item.used == 1 && current == index).then(|| *item)
    )
}
```

```
pub fn remove_item(&mut self, index: usize) -> Result<QueueItem,
ProgramError> {
    let (item_pos, mut item) = self.scan_items(
        |current, pos, _, item|
            (item.used == 1 && current == index).then(|| (pos, *item))
    )
    .ok_or(ProgramError::from(EphemeralVrfError::InvalidQueueIndex))?;

    item.used = 0;
    self.header.item_count = self.header.item_count.saturating_sub(1);
    let bytes =
        &mut self.acc[item_pos..item_pos + size_of::<QueueItem>()];
    Self::write_item_unaligned(bytes, &item);

    self.trim_trailing_holes();

    Ok(item)
}
```

```
pub fn find_item_by_id(&self, id: &[u8; 32]) -> Option<(usize,
QueueItem)> {
    self.scan_items(|current, _, _, item|
        (item.used == 1 && &item.id == id).then(|| (current, *item))
    )
}
```

C14: Unnecessary mixing of oracle insertion and removal logic

Severity: Minor

Status: Won't fix

Location

```
program/src/modify_oracles.rs  
    process_modify_oracles
```

Description

A single instruction `process_modify_oracles` is used both for adding or removing an oracle. Whether the operation is an insertion or a removal is determined by an `operation` argument which is matched against `0` or `1` ([link](#)).

Associating `0` with insertion and `1` with removal, together with the branching that follows, leads to unnecessary complexity, for example:

- We must fail if the `operation` argument does not match `0` or `1`.
- We must do custom conditioning on the operation type like:

```
if args.operation == 0  
    && (args.oracle_pubkey.0 == [0u8; 32]  
    || !validate_ristretto(&args.oracle_pubkey))
```

This complexity could be removed by simply separating the insertion and removal logic into separate instructions.

Recommendation

We suggest splitting `process_modify_oracles` into two separate instructions, one to insert an oracle and one to remove an oracle.

Resolution notes

The team decided to keep the single instruction to avoid modifying the instruction interface (discriminators, SDK, admin tooling).

C15: Redundant and complex request insertion logic

Severity: Minor

Status: Fixed

Location

`api/src/state/queue.rs`

Description

Each request is stored under an `id` that is a hash of several fields, one of which is the position where the item will be placed in the queue.

To know the position before the item is written, `process_request_randomness` first calls `insertion_position`, which walks the whole queue to work out where the next item would go ([link](#)). It then calls `add_item`, which walks the whole queue a second time to find that same position and write the item there ([link](#)).

This design is quite complex and fragile: inserting one request scans the entire queue twice, and the two scans must result in an identical inserting position.

Recommendation

In `process_request_randomness`, we suggest performing a single insertion scan. For this, we don't pre-compute the item's insertion index and combined hash. Instead, once the exact insertion byte is derived in `write_item_at`, we compute and set the combined hash (alongside the item's other fields which are set there).

Incidentally, we can delete the entire function `insertion_position`.

C16: Unnecessary use of unsafe Rust

Severity: **Minor**

Status: **Fixed**

Location

api/src/state/queue.rs

Description

Current behavior: In the *queue.rs* file, there are 4 conversions between raw bytes and typed values made with hand-written `unsafe` Rust, but these conversions could have been done using bytemuck safe helpers.

1) Reading the callback account metas as a typed slice ([link](#)):

```
// current
unsafe {
    from_raw_parts(bytes.as_ptr() as *const CompactAccountMeta,
        count)
}
// safe equivalent
bytemuck::cast_slice::<u8, CompactAccountMeta>(bytes)
```

2) Writing the account metas as bytes ([link](#)):

```
// current
unsafe {
    from_raw_parts(metas.as_ptr() as *const u8,
        metas_bytes_len)
}
// safe equivalent
bytemuck::cast_slice::<CompactAccountMeta, u8>(metas)
```

3) Reading one `QueueItem` out of a byte slice ([link](#)):

```
// current
unsafe { ptr::read_unaligned(bytes.as_ptr() as *const QueueItem) }
// safe equivalent
bytemuck::pod_read_unaligned::<QueueItem>(bytes)
```

4) Copying a `QueueItem` into a destination buffer ([link](#)):

```
// current
let src = unsafe {
    core::slice::from_raw_parts(item as *const QueueItem as *const
u8, size_of::<QueueItem>())
};
dst.copy_from_slice(src);
// safe equivalent
dst.copy_from_slice(bytemuck::bytes_of(item));
```

Expected behavior: Safe Rust (including `bytemuck` safe helpers) should be used whenever possible, since hand-written `unsafe` increases the risk of missing an invariant that Rust would otherwise have enforced.

Recommendation

We suggest replacing these four `unsafe` casts with their `bytemuck` equivalents (`cast_slice` , `pod_read_unaligned` , `bytes_of`), and adding a unit test that checks the safe and unsafe forms produce identical bytes.

C17: Misleading terminology and code formatting

Severity: Minor

Status: Fixed

Description

There are some misleading naming and readability aspects in the codebase, which reduce readability:

- 1) ([link](#)) — the local byte-offset variable `cursor` (recurring across the scan loops) shares its name with the field `header.cursor`. This name overshadowing makes the scan logic harder to follow.
- 2) ([link](#)) — the instruction enum is named `EphemeralVrfInstruction`, yet several variants (e.g. `Initialize`, `ModifyOracle`, `InitializeOracleQueue`, `CloseOracleQueue`) are not ephemeral-specific.
- 3) ([link](#)) — the field `open_queue: u64` is a count of queues but reads as a singular.
- 4) ([link](#)) — `solana_program::entrypoint!(process_instruction)` is at the bottom of the file after all the helpers it uses, despite being the top-level entrypoint to the program.
- 5) ([link](#)) — when initializing a queue PDA, the size increment is stored in the variable `realloc_size`. However, the PDA increment is not defined as `realloc_size`, but rather as `MAX_EXTRA_BYTES`, in the following specific situation:
 1. The PDA final size will exceed `MAX_EXTRA_BYTES` ([link](#)),
 2. The PDA is not yet owned by the VRF Program ([link](#)).It requires subtle reasoning to understand that in this case, `MAX_EXTRA_BYTES` and `realloc_size` are identical: (ii) guarantees that the PDA is currently empty while (i) ensures that the current increment is the maximal one, i.e. of `MAX_EXTRA_BYTES = 10_240 bytes`.
- 6) ([link](#)) — in `process_provide_randomness`, the local variable `let mut data = oracle_queue_info.try_borrow_mut_data()? shadows the function argument data: &[u8], so data refers to two different values in the same function.`

7) (link) — in `process_modify_oracles`, logging uses `msg!("Oracle has {} open queues", open_queue)`, which is inconsistent with other logs in this program, which are done through the `log` helper.

8) (link) — in `process_modify_oracles`, the borrowed `oracles_data` is released with an explicit `drop(oracles_data)`, which is inconsistent with the usual scoped block `{ ... }` used elsewhere in the codebase to release the same kind of borrow.

9) (link) — in `process_request_randomness`, the inserted item is initialized with field `used == 0`, but it will always be rewritten to `1` down the road in the helper `write_item_at`,

10) (link) — in `add_item`, the same variable `aligned` is unnecessarily defined twice.

Recommendation

We suggest fixing the above, for example as follows:

- 1) Renaming the local variable `cursor` to `offset_index`,
- 2) Renaming `EphemeralVrfInstruction` to `SolanaVrfInstruction`,
- 3) Renaming `open_queue` to `open_queues`,
- 4) Moving `solana_program::entrypoint!(process_instruction)` to the top of the file,
- 5) Always defining the PDA size increment as `realloc_size`,
- 6) Renaming the local variable `let mut data` to `raw_data`,
- 7) Always using the `log` helper for logging purposes,
- 8) Replacing the `drop` with a scoped block `{ ... }`,
- 9) Setting the new item's field `used` to `1` from the start, and not in `write_item_at`.
- 10) Defining the variable `aligned` once.

C18: Unnecessary arguments

Severity: Minor

Status: Fixed

Description

Some functions have unnecessary arguments:

- `create_pda` takes an `owner: &Pubkey` argument but it is always given the same value, i.e. this VRF program ID.
- `parse_instruction8` takes an `api_id: &Pubkey` argument but it is always given the same value, i.e. this VRF program ID.
- `align_up` takes an `align: usize` argument but it is always given the same value, i.e. the alignment of an item `align_of::<QueueItem>()`.
- `item_next` takes an `align: usize` argument but it is always given the same value, i.e. the alignment of an item `align_of::<QueueItem>()`.

Recommendation

In `create_pda`, `parse_instruction8`, `align_up` and `item_next`, we suggest removing the unnecessary argument, and instead fetching the relevant constant directly inside the function.

C19: Redundant stripping of the queue discriminator

Severity: Minor

Status: Fixed

Location

api/src/state/queue.rs
load

Description

Every queue instruction strips the 8-byte account discriminator (`&mut data[8..]`) before calling `QueueAccount::load` , for example in `process_close_oracle_queue` ([link](#)), while `load` itself documents this as the caller's responsibility ([link](#)).

However, repeating the offset at each call site is error-prone: a caller that forgets it, or slices the wrong offset, would misread the queue.

Recommendation

We suggest moving the discriminator handling inside `QueueAccount::load` so call sites pass the full account data and cannot get the offset wrong.

C20: Errors in comments

Severity: **Minor**

Status: **Fixed**

Description

Several comments contain information that does not match the implementation:

1) The "vrf-macro" terminology is erroneously used in various places. Here are a few examples (but an exhaustive list can be obtained by searching for "vrf-macro" in the codebase):

- ([link](#)) "vrf-macro an Oracle queue"; the intended word is "delegate" (leftover from a macro template).
- ([link](#)) "Call the delegation program to *vrf-macro* the Oracle queue account"; the intended word is "delegate".
- ([link](#)) "executes vrf-macro"; the intended word is "callback".
- ([link](#)) "Invoke the vrf-macro with the randomness"; the intended word is "callback".

2) ([link](#)) — both comments "*The owner program (account 2)*" and "*The delegation accounts (3-5)*" are wrong because in reality the delegation buffer is at index 2 and the owner program at index 6.

3) ([link](#)) — typo "mus" for "must" in "*The payer (account 0) mus be a signer.*"

4) ([link](#)) — the Oracle account is documented as a non-signer `[]`, but the handler enforces `oracle_info.is_signer()?`.

5) ([link](#)) — the function comment lists (payer at 0, queue at 1, buffer at 2, system at 3, with "*The payer (account 0) must be a signer*"), however:

- This does not match the code order `[oracle_queue_info, delegation_buffer, payer, system_program]` ([link](#)).
- There is no explicit signer check on the payer.

6) ([link](#)) — the function comment indexes two consecutive entries with the same number "2."

7) ([link](#)) — `oracle_queue_info` is documented as `[]` but the handler requires that it is writable via `is_writable()?`.

8) ([link](#)) — the comment about the inputs needed for the combined hash of a randomness request is missing the unix timestamp and the queue insertion index that the `hashv` call also includes ([link](#)).

9) ([link](#)) — the `metas_len` field comment says *"number of SerializableAccountMeta "*, but this region in fact stores a list of `CompactAccountMeta` , not `SerializableAccountMeta` .

10) ([link](#)) — typo "accrue" instead of "accrues" in *"// The oracle also accrue fees"*.

Recommendation

We suggest correcting these comments.

C21: Dead and unused code

Severity: Minor

Status: Fixed

Description

The codebase contains several pieces of unused logic:

- Unused error variants: the `EphemeralVrfError` variants `QueueFull`, `InvalidAccountData`, `AccountAlreadyInitialized`, `OracleNotRegistered`.
- The `instruction8!` macro carries `#[allow(unused_macros)]`, but the macro is used to generate serialization for 8 instruction structs.
- Unused binding in `process_request_randomness`: the index returned by `add_item` (link) is bound to a variable `_logical_index` that is never used.
- Unused off-chain helper `RequestRandomness::to_bytes`. Moreover, this helper hardcodes the `RequestHighPriorityRandomness` discriminant, although (i) there are 4 possible types of randomness request, and (ii) the default one should rather be `RequestHighPriorityRandomnessScoped`, as scoped requests are now the standard.

Recommendation

We suggest removing the unused error variants, the stale `#[allow(unused_macros)]` attribute, and the unused `_logical_index` binding, and deleting the unused `to_bytes` helper.

