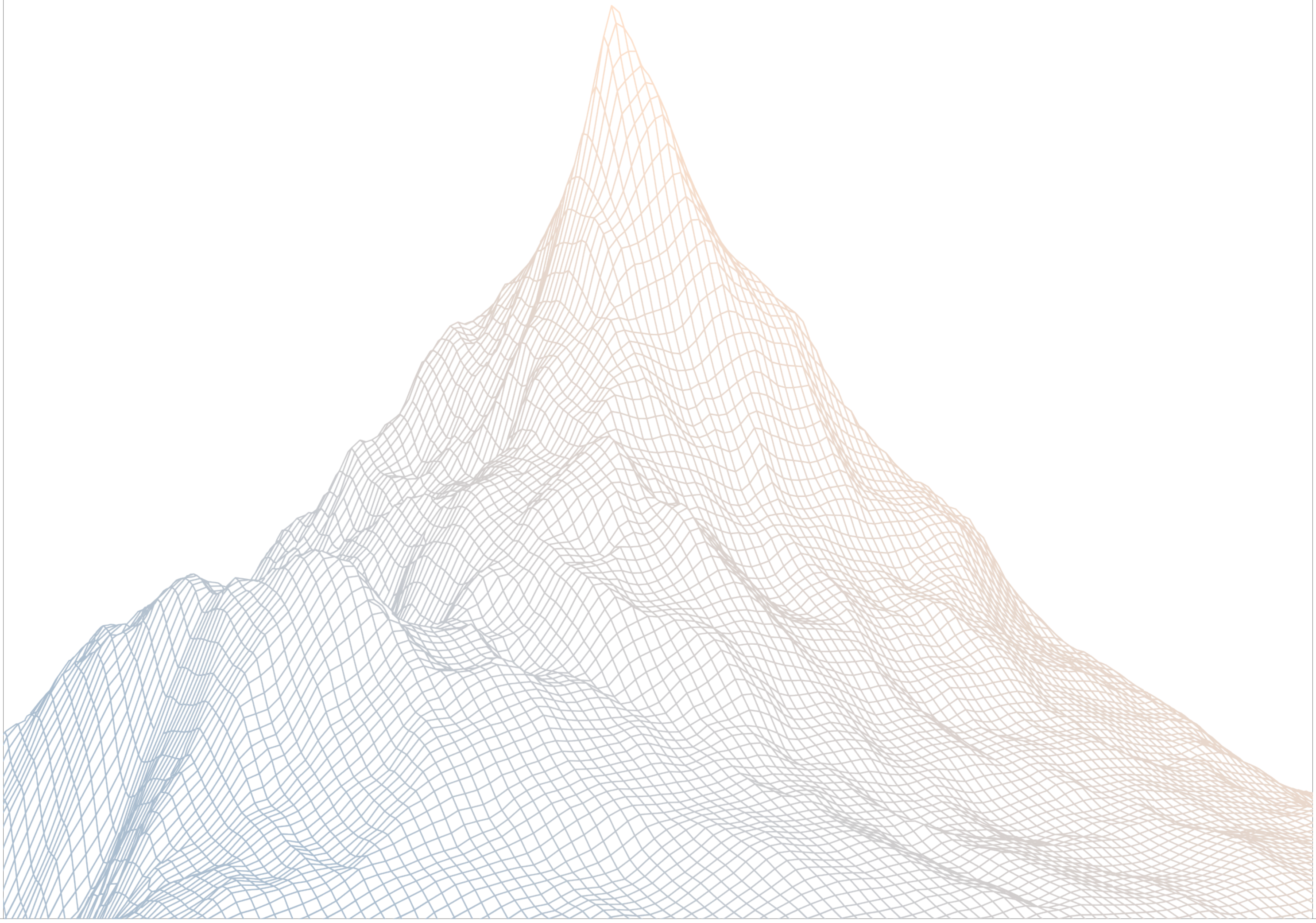
Zenith

MagicBlock

Smart Contract Security Assessment

VERSION 1.1



AUDIT DATES:	August 1st to August 6th, 2025
AUDITED BY:	0xhuy0512 Christian Vari

Contents

1 Introduction

1.1 About Zenith

Zenith assembles auditors with proven track records: finding critical vulnerabilities in public audit competitions.

Our audits are carried out by a curated team of the industry's top-performing security researchers, selected for your specific codebase, security needs, and budget.

Learn more about us at https://zenith.security.

1.2 Disclaimer

This report reflects an analysis conducted within a defined scope and time frame, based on provided materials and documentation. It does not encompass all possible vulnerabilities and should not be considered exhaustive.

The review and accompanying report are presented on an "as-is" and "as-available" basis, without any express or implied warranties.

Furthermore, this report neither endorses any specific project or team nor assures the complete security of the project.

1.3 Risk Classification

SEVERITY LEVEL	IMPACT: HIGH	IMPACT: MEDIUM	IMPACT: LOW
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

2 Executive Summary

2.1 About MagicBlock

MagicBlock builds next-gen blockchain infrastructure that scales applications without fragmentation. Our modular stack and ephemeral rollups enable high-speed, composable development for gaming, DeFi, and emerging Web3 applications on Solana.

2.2 Scope

The engagement involved a review of the following targets:

Target	ephemeral-vrf
Repository	https://github.com/magicblock-labs/ephemeral-vrf
Commit Hash	`75bc11cb0b616934207ad348db83c812ed3a0a4d`
Files	`api/*` `program/*`

2.3 Audit Timeline

August 1, 2025	Audit start
August 6, 2025	Audit end
August 14, 2025	Report published

2.4 Issues Found

SEVERITY	COUNT
Critical Risk	0
High Risk	0
Medium Risk	4
Low Risk	4
Informational	10
Total Issues	**18**

3 Findings Summary

ID	Description	Status
M-1	Predictable random numbers due to hash reuse	Resolved
M-2	Denial of service by filling queue with invalid requests	Resolved
M-3	Fee theft via premature queue closure	Resolved
M-4	DoS and economic drain via malicious callback registration	Resolved
L-1	Oracles removed by admin cannot close their queue to re-claim lamports	Resolved
L-2	Loss of service from premature oracle removal	Resolved
L-3	Inconsistent signer flag in close_oracle_queue SDK call	Resolved
L-4	Account spoofing in process_undelegate_oracle_queue	Resolved
I-1	Inconsistent instruction discriminator length	Resolved
I-2	Unused field in instruction arguments	Resolved
I-3	Unused enum variant	Resolved
I-4	Misleading variable name vrf_program_data	Resolved
I-5	Inconsistent program ID reference leads to reduced code clarity	Resolved
I-6	Inconsistent System Program Address Validation	Resolved
I-7	Redundant Check on Oracle Registration Slot	Resolved
I-8	Redundant lamport check in is_empty_or_zeroed	Resolved
I-9	Missing oracle identity validation permits invalid entries	Resolved
I-10	Inconsistent PDA bump handling	Resolved

4 Findings

4.1 Medium Risk

A total of 4 medium risk findings were identified.

[M-1] Predictable random numbers due to hash reuse

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Low

Target

- program/src/request_randomness.rs#L72-L80

Description:

A user can request a random number, have an oracle fulfill it, and then submit another request within the same Solana block. While this scenario depends on an oracle reacting quickly to the request, it is not impossible for this to occur. When using MagicBlock's ephemeral rollup for Queue accounts, where one Solana block is divided into multiple smaller blocks, the likelihood of this scenario increases. Because the `slot`, `slothashes`, and `timestamp` are identical for both requests, the `combined_hash` can be reused, which leads to the same random number being produced. This allows an attacker to predict the outcome and exploit any logic that relies on this randomness for their benefit.

```
let combined_hash = hashv(&[
    &args.caller_seed,
    &slot.to_le_bytes(),
    &slothash,
    &args.callback_discriminator,
    &args.callback_program_id.to_bytes(),
    &time.to_le_bytes(),
    &idx.to_le_bytes(),
]);
```

Recommendations:

The `QueueItem` struct already contains a `slot` field which should be used to store the slot in which the request was created. The fulfillment logic must then ensure that the randomness is provided in a subsequent slot.

```
pub fn process_provide_randomness(accounts: &[AccountInfo<'_>], data:
    &[u8]) -> ProgramResult {
     ...
    if Clock::get()?.slot ≤ item.slot {

     return Err(ProgramError::from(VrfError::OracleMustProvideInDifferentSl
            ot).into());
    }
     ...
}
```

Magicblock: Resolved with @0a1d1475120....

Zenith: Verified

[M-2] Denial of service by filling queue with invalid requests

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Low

Target

- src/provide_randomness.rs#L82-L88
- src/provide_randomness.rs#L126

Description:

A random number requester can craft a request that is guaranteed to fail during fulfillment. There are two ways to achieve this:

1. Make `item.callback_accounts_meta` include the `oracle_info.key` address, which leads to a revert with `InvalidCallbackAccounts`.
2. Include a callback CPI that is structured to always fail, causing the random fulfillment transaction to panic.

When a queue item is not fulfilled, it remains in the queue, occupying one of the 25 available slots. These failed items can only be deleted when an Oracle closes the entire queue using the `close_oracle_queue()` instruction.

An attacker can exploit this by injecting multiple invalid random requests, filling up the queue and displacing legitimate requests. For instance, an attacker could fill 23 slots with invalid requests, leaving only two slots for legitimate use. If those two slots are occupied by valid, unfulfilled requests, any new legitimate requests will be rejected. The cost for such an attack is minimal, as the cost to request 23 random words is: `500_000 * 23 = 11_500_000 = 0.011 SOL ~ 1.5$`

Recommendations:

Implement a method that allows the Oracle to manually delete faulty request items from the queue. This would allow clearing malicious entries without disrupting the service for legitimate users by closing the whole queue.

Magicblock: Resolved with @e5d06b68b04...

Zenith: Verified

[M-3] Fee theft via premature queue closure

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Medium

Target

- program/src/close_oracle_queue.rs

Description:

The `close_oracle_queue` instruction allows oracles to prematurely close queues and extract accumulated user fees without fulfilling the associated randomness requests.

When a user submits a randomness request, they are charged `500,000` or `800,000` lamports, which are transferred directly to the oracle queue PDA using `system_instruction::transfer` in `request_randomness.rs`. These fees are intended as payment for randomness services, to be collected by oracles only after successfully fulfilling the request via `provide_randomness`.

However, the `close_oracle_queue` function lacks any validation to ensure that the queue has been fully processed before it is closed. This oversight enables an oracle to invoke `close_account(oracle_queue_info, oracle_info)` while the queue still contains pending requests. When executed, this function transfers all lamports from the queue PDA to the oracle's personal account, regardless of whether any requests have been fulfilled.

As a result, the oracle can effectively steal user payments while abandoning the associated randomness requests, which are then permanently lost. This breaks the core service guarantee of the VRF system and allows oracles to extract unearned revenue.

Recommendations:

We recommend preventing queue closure if there are any unprocessed requests or refund fees to the legit users.

Magicblock: Resolved with @cccf967ac16....

Zenith: Verified.

[M-4] DoS and economic drain via malicious callback registration

`SEVERITY:` Medium	`IMPACT:` Medium
`STATUS:` Resolved	`LIKELIHOOD:` Medium

Target

- program/src/request_randomness.rs
- vrf-oracle/src/oracle/processor.rs

Description:

The request queue is vulnerable to a dual-impact denial-of-service (DoS) and economic drain attack, where an adversary can permanently fill the oracle queue with requests containing malicious or invalid callbacks. These poisoned entries prevent legitimate users from submitting new randomness requests and simultaneously force oracles to process failing transactions at a financial loss.

The vulnerability lies in the `request_randomness` flow, where users are permitted to specify arbitrary `callback_program_id` and `callback_accounts_metas`. An attacker can exploit this by registering callbacks designed to always fail, such as programs that unconditionally panic, use invalid account configurations, exceed compute limits, or reference non-existent program IDs.

When oracles process these entries in `process_oracle_queue`, they generate valid VRF proofs and attempt to invoke the callbacks using Solana's CPI mechanism. If the callback fails, Solana's transaction model reverts the entire operation. As a result, the item is not removed and will be retried indefinitely.

This behavior leads to two severe consequences:

1. Denial-of-Service: With a cost of just `500,000` lamports per entry (~$0.08), an attacker can saturate the 25-slot queue for approximately `$2.10`. Once full, legitimate programs are unable to enqueue new requests, effectively locking them out of the randomness service.
2. Economic Damage: Each failed attempt burns `20,000` lamports. The vrf-oracle's retry logic allows up to 5 retries per item per trigger, and since queue updates retrigger processing of all entries, the same malicious items can be retried hundreds or thousands of times. Oracles thus operate at a consistent loss, potentially incurring daily costs in the hundreds or thousands of dollars, especially under sustained attack.

With no failure tracking, blacklisting, or circuit-breaker logic in place, this vulnerability allows low-cost, persistent attacks that exhaust system resources and financially punish oracle operators while rendering the randomness service unavailable to honest users.

Recommendations:

We recommend implementing the following safeguards:

- Introduce a time-to-live (TTL) for each randomness request. Entries that exceed their TTL without successful execution should automatically expire and be purged from the queue.
- Introduce a dynamic fee mechanism.
- Integrate pre-processing simulation in the vrf-oracle codebase. This would allow the oracle to simulate the callback in advance (e.g., using dry-run logic) and detect failure conditions before incurring CPI costs.

Magicblock: Resolved with @e5d06b68b04... and @9db7b55ba9d.... Fees are kept static as it's a business requirement to keep them fixed and predictable.

Zenith: Verified.

4.2 Low Risk

A total of 4 low risk findings were identified.

[L-1] Oracles removed by admin cannot close their queue to reclaim lamports

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Target

- src/close_oracle_queue.rs#L40-L45

Description:

The `close_oracle_queue()` instruction is intended to allow oracles to close their associated queue account and reclaim the lamports. However, the instruction validates the existence of the oracle's data account.

If the admin removes an oracle from the system using the `ModifyOracle` function, the oracle's data account is deleted. Consequently, when the oracle operator attempts to call `close_oracle_queue()`, the call will fail because the check for the `oracle_data_info` account will no longer pass. This locks the lamports in the queue account, preventing the oracle operator from retrieving their funds.

Recommendations:

The validation for the oracle's data account should be removed from the `close_oracle_queue()` instruction. This will allow oracle operators to close their queue and reclaim their lamports even after being removed from the system by an administrator.

Magicblock: Resolved with @cccf967ac16....

Zenith: Verified.

[L-2] Loss of service from premature oracle removal

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Target

- program/src/modify_oracles.rs

Description:

The `modify_oracles instruction` allows the removal of an oracle without verifying whether it still has pending user obligations. When called with `operation == 1`, the instruction removes the specified oracle and immediately closes its associated `oracle_data account`, without checking for existing oracle queues or in-flight randomness requests.

This presents a serious service integrity issue. Each oracle queue is tied to a specific oracle through the PDA seed structure `[QUEUE, oracle_key, index]`. Once its `oracle_data` account is closed, any call to `provide_randomness` fails when attempting to load the oracle's data. The consequence is that all randomness requests in the removed oracle's queues become permanently unfulfillable, even though users have already paid for them.

This results in direct financial loss for affected users and irrecoverable denial of service. There is no fallback or reassignment mechanism in place, and no other oracle can step in to process the abandoned requests due to PDA constraints.

Recommendations:

We recommend implementing a validation check before removing an oracle to ensure it has no active queues or pending randomness requests. The system should prevent removal if the oracle is still responsible for any outstanding user obligations.

Alternatively, we recommend introducing a fee recovery or reassignment mechanism for queues tied to removed oracles to protect user funds in edge cases.

Magicblock: Resolved with @dc6425728b0... and @9db7b55ba9d....

Zenith: Verified.

[L-3] Inconsistent signer flag in `close_oracle_queue` SDK call

`SEVERITY:` Low	`IMPACT:` Low
`STATUS:` Resolved	`LIKELIHOOD:` Low

Target

- api/src/sdk.rs

Description:

The `close_oracle_queue` function in constructs an `AccountMeta` for the oracle identity with `is_signer` set to `false`. However, the corresponding instruction handler enforces that the oracle account must be a signer via `oracle_info.is_signer()?`. This creates a mismatch between the SDK interface and the actual on-chain signing requirement.

The SDK currently implies that the oracle does not need to sign the transaction, which is incorrect.

Recommendations:

We recommend updating the `AccountMeta` for the oracle identity with `is_signer` set to `true`.

Magicblock: Resolved with @cccf967ac16....

Zenith: Verified.

[L-4] Account spoofing in `process_undelegate_oracle_queue`

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Target

- program/src/undelegate_oracle_queue.rs

Description:

The `process_undelegate_oracle_queue` instruction contains a validation bypass vulnerability in its handling of the Magic program and context accounts. The function accepts `magic_program` and `magic_context` as input accounts (lines 48-49) but fails to validate that they match the expected `MAGIC_PROGRAM_ID` and `MAGIC_CONTEXT_ID` constants.

This creates a correctness flaw because the `commit_and_undelegate_accounts` function ultimately constructs a CPI instruction using `magic_program.key` as the target program ID (line 53 in ephem.rs), meaning any program passed by the caller will be invoked instead of the legitimate Magic program.

The missing `magic_context` validation compounds this issue by allowing to provide incorrect context data that could further manipulate the behavior of whatever program is substituted.

The issue is reported with Low severity since this instruction can be executed only by the authority of the delegated queue.

Recommendations:

We recommend adding strict validation checks to enforce that:

- `magic_program.key` equals the expected `MAGIC_PROGRAM_ID`
- `magic_context.key` equals the expected `MAGIC_CONTEXT_ID`

Magicblock: Resolved with @f70c732b052....

Zenith: Verified.

4.3 Informational

A total of 10 informational findings were identified.

[I-1] Inconsistent instruction discriminator length

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- program/src/lib.rs#L30

Description:

The `process_instruction` function in `program/src/lib.rs` uses a single byte as the instruction discriminator (`ix`). This is inconsistent with other parts of the program, as the `ProcessUndelegation` instruction discriminator and all account discriminators are 8 bytes long. This inconsistency in data handling can lead to maintainability issues and potential bugs.

Recommendations:

It is recommended to use a consistent discriminator length across all instructions. The instruction discriminator should be standardized to 8 bytes to match the length of account discriminators and align with best practices, improving code clarity and reducing the chance of errors.

Magicblock: Resolved with @1bea7edf461....

Zenith: Verified.

[I-2] Unused field in instruction arguments

`SEVERITY:` Informational	`IMPACT:` Informational
`STATUS:` Resolved	`LIKELIHOOD:` Low

Target

- program/src/provide_randomness.rs#L31

Description:

The `oracle_identity` field in `ProvideRandomness` at `program/src/provide_randomness.rs` is unused.

Recommendations:

Remove the unused argument to improve code clarity.

Magicblock: Resolved with @97bcdd8efca....

Zenith: Verified.

[I-3] Unused enum variant

`SEVERITY:` Informational	`IMPACT:` Informational
`STATUS:` Resolved	`LIKELIHOOD:` Low

Target

- api/src/state/mod.rs#L20

Description:

The `AccountDiscriminator` enum at `api/src/state/mod.rs` defines a `Counter` variant that is never used.

Recommendations:

Remove the unused `Counter` variant to improve code clarity.

Magicblock: Resolved with @97bcdd8efca....

Zenith: Verified.

[I-4] Misleading variable name `vrf_program_data`

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- src/modify_oracles.rs#L37

Description:

The `vrf_program_data` variable at `src/modify_oracles.rs` is misleadingly named. It represents the BPF upgradeable program's data account and is used only to fetch the program's upgrade authority, not for VRF-specific data. This could cause confusion regarding its true purpose.

Recommendations:

Rename the variable to better reflect its generic nature.

Magicblock: Resolved with @c8a01ad1a4e....

Zenith: Verified.

[I-5] Inconsistent program ID reference leads to reduced code clarity

`SEVERITY:` Informational	`IMPACT:` Informational
`STATUS:` Resolved	`LIKELIHOOD:` Low

Target

- program/src/request_randomness.rs#L61

Description:

The codebase inconsistently uses `ephemeral_vrf_api::ID` and `ID` to refer to the same program ID. This can mislead developers into thinking two different programs are involved.

Recommendations:

Enforce a single, consistent alias for the program ID throughout the codebase.

Magicblock: Resolved with @16c77af0ff4....

Zenith: Verified.

[I-6] Inconsistent System Program Address Validation

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- program/src/initialize.rs#L30

Description:

The `process_initialize` function in `program/src/initialize.rs` validates the `system_program` account. This check is redundant because any Cross-Program Invocation (CPI) to an incorrect system program address will cause the transaction to fail regardless. Other instructions in the codebase already omit this unnecessary check

Recommendations:

For gas savings and code consistency, remove the unnecessary validation from the `process_initialize` function.

Magicblock: Resolved with @16c77af0ff4....

Zenith: Verified.

[I-7] Redundant Check on Oracle Registration Slot

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- src/initialize_oracle_queue.rs#L78-L84

Description:

In `initialize_oracle_queue.rs`, there is a check to validate that an oracle has been registered for at least one slot before it can be added to a queue. The check `if slots_since_registration == 0` is redundant. The scenario where `current_slot` is less than `oracle_data.registration_slot` is not possible outside of a test environment. Moreover, a subsequent check, `if slots_since_registration < 200`, already provides a sufficient safeguard against using a newly registered oracle immediately. Removing the explicit zero-slot check simplifies the logic without compromising security.

Recommendations:

It is recommended to remove the unnecessary check for `slots_since_registration == 0`.

Magicblock: Resolved with @1844400b9ce....

Zenith: Verified.

[I-8] Redundant lamport check in `is_empty_or_zeroed`

`SEVERITY`: Informational	`IMPACT`: Informational
`STATUS`: Resolved	`LIKELIHOOD`: Low

Target

- api/src/loaders.rs#L52

Description:

The `is_empty_or_zeroed` function in `api/src/loaders.rs` currently verifies if an account is uninitialized by checking if its data is zeroed or its lamport balance is zero. The expression used is `data.iter().all(|&b| b == 0) || lamports == 0`.

The `lamports == 0` part of the check is redundant. If an account has data, it must have a non-zero lamport balance to be rent-exempt, rendering the lamport check ineffective. If the account data is empty, `data.iter().all()` correctly evaluates to `true`, making the lamport check unnecessary.

Recommendations:

To optimize gas and improve clarity, the redundant lamport check should be removed.

```diff
- let is_zeroed = data.iter().all(|&b| b == 0) || lamports == 0;
+ let is_zeroed = data.iter().all(|&b| b == 0);
```

Magicblock: Resolved with @1844400b9ce....

Zenith: Verified.

[I-9] Missing oracle identity validation permits invalid entries

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- program/src/modify_oracles.rs

Description:

The `process_modify_oracles` function fails to enforce validation on the `args.identity` field, allowing any arbitrary `Pubkey` to be added as an oracle identity. At line 69, the identity is directly appended to the oracle list using `oracles.oracles.push(args.identity)`, and at line 62 it is used to derive a PDA without any semantic validation.

This oversight enables problematic values, such as `Pubkey::default()` to be incorrectly registered as oracle identities. These entries may introduce inconsistencies across the system, as Pubkey::default() is commonly interpreted as an “uninitialized” or “null” value, and system addresses are not representative of real oracle operators. Inclusion of such keys could disrupt assumptions in oracle processing logic, PDA derivation, or administrative tooling.

Recommendations:

We recommend implementing strict validation for `args.identity` before inclusion.

Magicblock: Resolved with @f70c732b052....

Zenith: Verified

[I-10] Inconsistent PDA bump handling

`SEVERITY:` Informational	`IMPACT:` Informational
`STATUS:` Resolved	`LIKELIHOOD:` Low

Target

- program/src/initialize.rs

Description:

An inconsistency exists in the `process_initialize()` function regarding the handling of Program Derived Addresses (PDAs).

The validation logic at line 29 employs dynamic PDA derivation using `oracles_info.has_seeds(&[ORACLES], &ephemeral_vrf_api::ID)?`, which internally `calls Pubkey::find_program_address()` to compute the correct PDA address and bump value based on runtime conditions, including the actual deployed program ID. This ensures accurate verification of the oracles_info account.

However, the creation logic at lines 35—43 diverges by invoking `create_pda()` with a hardcoded bump value of `255`, returned from `oracles_pda().1`. This bypasses the dynamic derivation process and risks generating an incorrect PDA.

This discrepancy introduces a fragility: if the true bump for [ORACLES] with the current program ID is not `255`, the validation step will succeed, but the subsequent PDA creation would be incorrect and could use a public key that is a legit point of the elliptic curve.

Additionally, the `create_pda()` function itself accepts the PDA `seeds`, `bump`, and `owner` as parameters alongside the `target_account.key`, but does not internally verify that these inputs produce the claimed PDA address. The absence of a validation means that the function can be invoked with mismatched inputs, for example a valid PDA address but incorrect seed/bump/owner parameters, or vice versa.

Recommendations:

We recommend eliminating the use of hardcoded bump values and instead retrieving the PDA and bump dynamically.

Magicblock: Resolved with @f70c732b052....

Zenith: Verified