Jupiter Perp

Audit

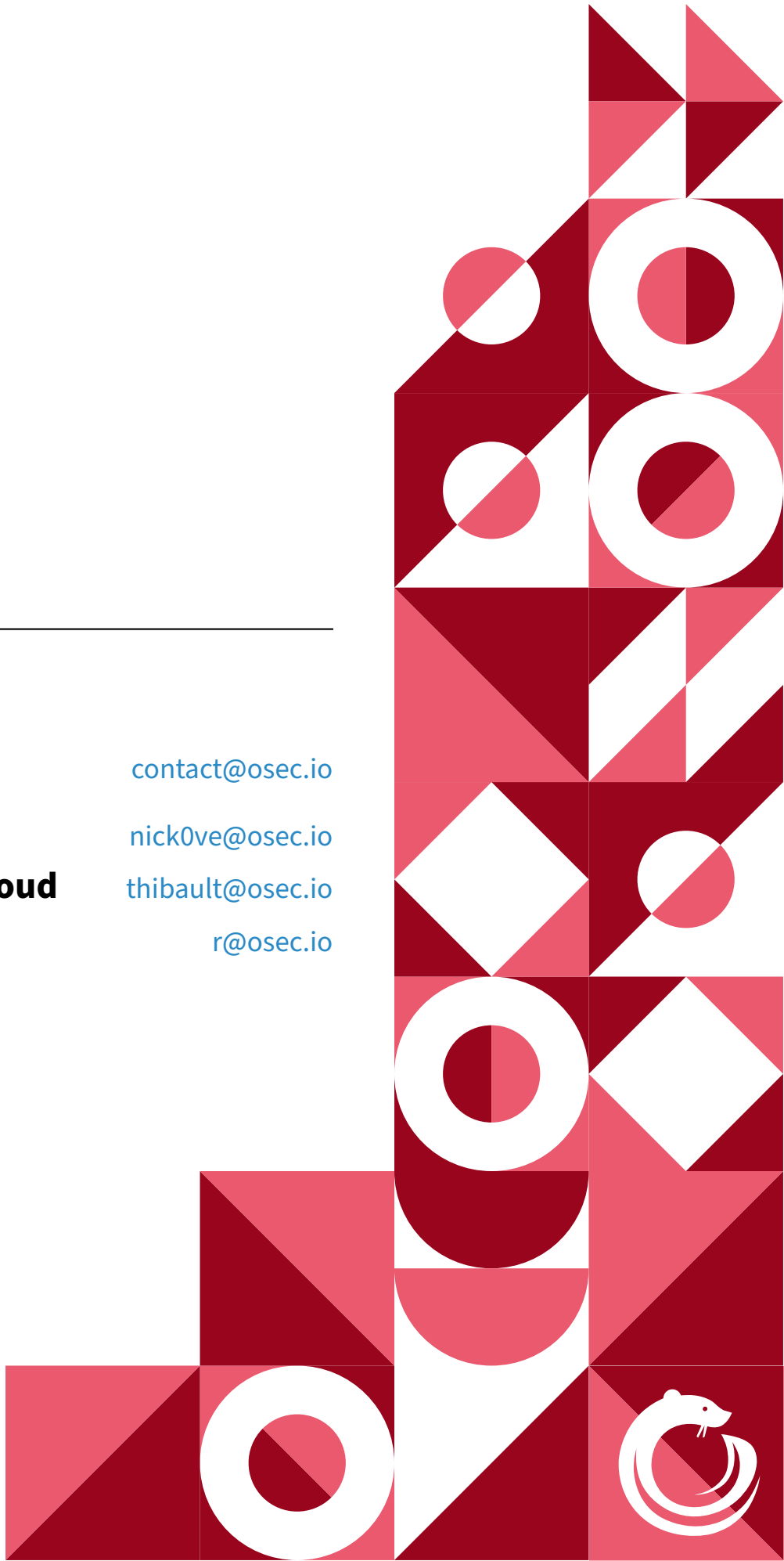
Presented by:

OtterSec contact@osec.io

Nicola Vella nick0ve@osec.io

Thibault Marboud thibault@osec.io

Robert Chen r@osec.io



Contents

01 | Executive Summary

Overview

Jupiter engaged OtterSec to perform an assessment of the `perpetuals` program. This assessment was conducted between October 3rd and November 15th, 2023. For more information on our auditing methodology, see Appendix B.

Key Findings

Over the course of this audit engagement, we produced 9 findings in total.

In particular, we have identified two high-risk issues, including a rounding error in the computation of the average position price (OS-JPT-ADV-00) and the possibility of updating a position request to front-run the Keeper (OS-JPT-ADV-01).

Furthermore, we have highlighted a fund loss issue in the case of a malicious keeper (OS-JPT-ADV-03). and an event manipulation issue (OS-JPT-ADV-06).

We also made a recommendation around the lack of multiple pool authorities and inclusion of a predefined withdraw delay, potentially raising security issues in the future (OS-JPT-SUG-01) and suggested including a length constraint on the pool name to prevent collisions with pool accounts (OS-JPT-SUG-00).

02 | Scope

The source code was delivered to us in a git repository at github.com/jup-ag/perpetuals. This audit was performed against commit 33acf0e.

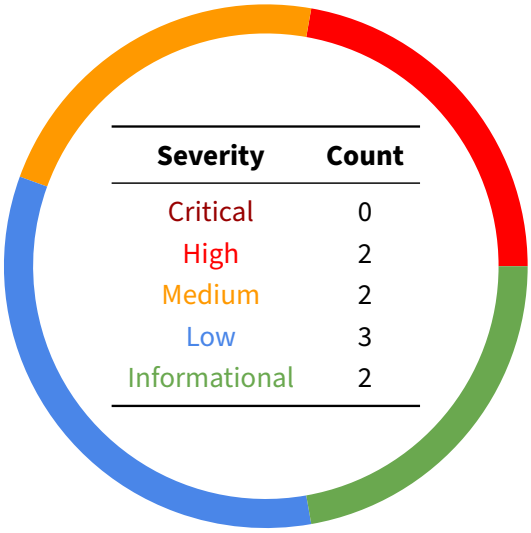
A brief description of the programs is as follows:

Name	Description
perpetuals	An open-source implementation of a non-custodial decentralized exchange supporting leveraged trading across a range of assets.

03 | Findings

Overall, we reported 9 findings.

We split the findings into **vulnerabilities** and **general findings**. Vulnerabilities have an immediate impact and should be remediated as soon as possible. General findings do not have an immediate impact but will aid in mitigating future vulnerabilities.



Severity	Count
Critical	0
High	2
Medium	2
Low	3
Informational	2

04 | Vulnerabilities

Here, we present a technical analysis of the vulnerabilities we identified during our audit. These vulnerabilities have *immediate* security implications, and we recommend remediation as soon as possible.

Rating criteria can be found in Appendix A.

ID	Severity	Status	Description
OS-JPT-ADV-00	High	Resolved	Rounding down `next_price` results in the constant average price during minor increments of `size_usd_delta`.
OS-JPT-ADV-01	High	Resolved	Initiating `update_decrease_position_request` and `update_increase_position_request` with the intent to front-run the keeper.
OS-JPT-ADV-02	Medium	Resolved	Integer overflow in `swap_usd_amount` and integer underflow in `dispensing_custody` due to a lack of checks in swap instructions.
OS-JPT-ADV-03	Medium	Resolved	An incorrect check for program ID allows a keeper to drain a pool's `collateral_custody_token_account`.
OS-JPT-ADV-04	Low	Resolved	Utilizing `SystemAccount` for ownership validation may prevent the closure of open positions.
OS-JPT-ADV-05	Low	Resolved	`Custody::validate` fails to call `self.funding_rate_state.validate`, allowing the setting of the funding rate to high values.
OS-JPT-ADV-06	Low	Resolved	The `amount` field in `ClosePositionRequestEvent` may display incorrect information.

OS-JPT-ADV-00 [high]| Rounding Error

Description

The `size_usd_delta` parameter in `get_new_price` is manipulable to artificially increase the position's average price without a corresponding increase in the average position price. `get_new_price` calculates the new average price of a trading position based on changes in size and the current market price. The vulnerability arises since the function does not enforce a minimum `size_usd_delta`, allowing an attacker to incrementally increase the position size while keeping the average price identical.

get_position.rs RUST

```rust
pub fn get_new_price(&self, next_price: u64, size_usd_delta: u64) -> Result<u64> {
    if self.size_usd == 0 {
        return Ok(next_price);
    }

    if size_usd_delta == 0 {
        return Ok(self.price);
    }

    let next_size = math::checked_add(self.size_usd, size_usd_delta)?;
    // get PnL delta
    let price_delta = self.price.abs_diff(next_price);
    let pnl_delta: u64 = math::checked_as_u64(math::checked_div(
        math::checked_mul(self.size_usd as u128, price_delta as u128)?,
        self.price as u128,
    )?)?;

    // when price go up, pnl is positive for long, negative for short
    // when price go down, pnl is negative for long, positive for short
    let next_size_with_pnl = if next_price > self.price {
        math::checked_add(next_size, pnl_delta)?
    } else {
        math::checked_sub(next_size, pnl_delta)?
    };

    math::checked_as_u64(math::checked_div(
        math::checked_mul(next_size as u128, next_price as u128)?,
        next_size_with_pnl as u128,
    )?)
}
```

This results in the average price for a position not rising proportionally and remaining consistent. Consequently, given that the position price is now notably higher, while the average price remains unchanged, the user realizes a substantial profit upon exiting the position, as the protocol utilizes the average price to compute the profit and loss (PnL) during the exit.

Proof of Concept

1. Assume there is a long position with an initial `size_usd` quantity and an initial `average_price`.
2. The market price increases.
3. An attacker incrementally increases the position with a small `size_usd_delta` (e.g., one) each time.
4. Since the protocol utilizes round-down logic when calculating the average price, the small `size_usd_delta` does not trigger any change, and the average price effectively remains the same.
5. After multiple iterations of increasing the position size with small `size_usd_delta` values, the position's `size_usd` becomes larger, but the average price remains the same.
6. Upon the attacker's decision to close the position, the calculation of profit occurs utilizing the average price. Given that the average price has not risen in proportion to the growth in position size, the attacker realizes a profit from the position.

Remediation

Ensure for long positions, the average price is rounded up, while for short positions, the average price is rounded down.

Patch

Fixed in 80fdf99.

OS-JPT-ADV-01 [high]| Front-Running Position Execution

Description

`update_decrease_position_request` and `update_increase_position_request` instructions allow users to update their position requests at anytime before the actual execution. The user can use his advantage to front-runs the keeper by adjusting their position based on the latest oracle update. A user may exploit this to gain an advantage by adjusting their position based on market movements after setting the trigger.

perpetuals/src/lib.rs RUST

```rust
pub mod perpetuals {
    use super::*;
    // admin instructions
    pub fn update_increase_position_request(
        ctx: Context<UpdateIncreasePositionRequest>,
        params: UpdateIncreasePositionRequestParams,
    ) -> Result<()> {
        instructions::update_increase_position_request(ctx, &params)
    }

    pub fn update_decrease_position_request(
        ctx: Context<UpdateDecreasePositionRequest>,
        params: UpdateDecreasePositionRequestParams,
    ) -> Result<()> {
        instructions::update_decrease_position_request(ctx, &params)
    }
    [...]
}
```

Thus, by updating the position request, the user essentially front-runs the keeper by adjusting their position based on the latest market conditions slightly before the keeper executes the request. The user may increase the leverage to secure a more advantageous PnL in a favorable market movement. Conversely, they may decrease leverage for a reduced position size to mitigate losses in an unfavorable market movement.

Remediation

Ensure that the execution of a request considers modification of the request within the same slot. Store the slot information of the request inside the corresponding account and check it during the execution process.

Patch

Fixed in a1e35e7.

OS-JPT-ADV-02 [med]| Integer Overflow/Underflow

Description

The vulnerability pertains to a potential integer overflow that may occur in all swap instructions during the computation of `swap_usd_amount` when handling high-valued assets. Specifically, `swap_usd_amount`, being of type `u64`, is susceptible to overflow if the product of the received token's price and the input amount surpasses the maximum value representable by a `u64`.

perpetuals/src/instruction/swap_exact_out.rs RUST

```rust
let swap_usd_amount = math::checked_mul(
    received_token_price
        .scale_to_exponent(-(Perpetuals::PRICE_DECIMALS as i32))?
        .price,
    params.amount_in.0,
)?;
```

An overflow of this nature may yield inaccuracies in calculating USD values for the swap, consequently affecting fee calculations and impacting the overall functionality of the swapping logic. Furthermore, the swap functions lack validation to ensure that the `dispensing_custody` account possesses adequate liquidity before executing the swap. The `dispensing_custody` account, responsible for providing funds for the `amount_out` value, may thus experience an underflow if insufficient funds exist.

Proof of Concept

Consider an example using Bitcoin (BTC) as the received token with a very high price:

RUST

```rust
// BTC oracle price
let received_token_price = OraclePrice {
    price: 34918115102500 // 34918.1151025 USD
    exponent: -8,  // 10^(-8)
};

// Input amount in SOL
let amount_in = 1000000000;  // 1 SOL
```

As shown above, `price` is 34918115102500 and `amount_in` is 1000000000, thus `swap_usd_amount` will be `price*amount_in`, which results in a value that is greater than `U64:max`, overflowing.

Remediation

Ensure to check for potential overflow scenarios or modify the type of `swap_usd_amount` amount to handle larger numbers without any issues. Additionally, validate that `dispensing_custody` account contains enough funds to cover the intended swap.

Patch

Fixed in 224a273.

OS-JPT-ADV-03 [med]| Fund Loss Via Malicious Keeper

Description

There is an incorrect check in `increase_position_pre_swap` during the verification of the program ID. The validation checks whether the program ID linked to the current instruction (`current_ixn.program_id`) aligns with the anticipated program ID (`*ctx.program_id`). The issue stems from utilizing the program ID from the current instruction instead of the program ID associated with the `increase_position_ixn` instruction.

increase_position_pre_swap.rs RUST

```rust
pub fn increase_position_pre_swap(
    ctx: Context<IncreasePositionPreSwap>,
    _params: &IncreasePositionPreSwapParams,
) -> Result<()> {
    [...]
    // Check Increase Position Ix
    if let Ok(increase_position_ixn) = load_instruction_at_checked(current_idx + 2,
        ↪  &instruction) {
        require_keys_eq!(
            current_ixn.program_id,
            *ctx.program_id,
            PerpetualsError::CPINotAllowed
        );
    [...]
}
```

This introduces a vulnerability where a malicious actor, the keeper, may exploit the system by draining the `collateral_custody_token_account`.

Proof of Concept

1. The vault keeper initiates a valid transaction (`JupiterPerps::increase_position_pre_swap`) involving `PositionRequest1`, with `PositionRequest1.pre_swap_amount` and `collateral_custody_token_account.amount` both set to 10,000.
2. The system executes the `Jupiter::shared_accounts_route` instruction, which swaps some tokens through Jupiter, resulting in an increase in `collateral_custody_token_account.amount` to 10,100.
3. The malicious keeper sends a padding instruction to a program they control. This instruction does nothing except satisfy all the checks enforced by `increase_position_pre_swap`. The keeper waits for legitimate transactions that will increase `collateral_custody_token_account.amount` to, for example, 100,000.

4. The malicious keeper sends another transaction (`JupiterPerps::increase_position`) claiming to increase `PositionRequest1`. The vulnerable code mistakenly calculates the deposited amount based on the previous `collateral_custody_token_account.amount` and the initial `PositionRequest1.pre_swap_amount`, by subtracting them to obtain 89900, while the keeper deposited only 100. This allows the attacker to extract tokens without the system realizing the discrepancy.

Remediation

Utilize the correct program ID (`increase_position_ixn.program_id`) in the check. This ensures the check validates the program ID associated with the increased position instruction.

Patch

Fixed in ac98728.

OS-JPT-ADV-04 [low] | Inability To Close Position

Description

`ClosePositionRequest` allows the owner or a whitelisted keeper to close a perpetual position request. It transfers tokens and closes accounts associated with the position request, emitting an event to capture the relevant information.

close_position_request.rs RUST

```rust
pub struct ClosePositionRequest<'info> {
    pub keeper: Option<Signer<'info>>,
    #[account(mut)]
    pub owner: SystemAccount<'info>,
    [...]
}
```

The issue is related to the ownership of the position account when created by a program derived address owned by another program. In `ClosePositionRequest`, the protocol assumes the owner of the position to be a `SystemAccount`. However, a position could be created by an external program using a program derived address (PDA). In that case, the owner account validation will fail due to the `SystemAccount` type, thus rendering this instruction unusable, preventing the closure of open positions.

Remediation

Replace the `SystemAccount` type with `UncheckedAccount`, enabling program derived addresses to invoke this instruction.

Patch

Fixed in 802da52.

OS-JPT-ADV-05 [low] | Missing Validation

Description

`validate` in `Custody` performs validation checks on the parameters and attributes of a custody account. It returns a boolean value indicating whether the custody is valid based on specific criteria.

custody.rs RUST

```rust
impl FundingRateState {
    pub fn validate(&self) -> bool {
        (self.hourly_funding_bps as u128) <= Perpetuals::BPS_POWER
    }
}

impl Custody {

    pub fn validate(&self) -> bool {
        self.token_account != Pubkey::default()
            && self.mint != Pubkey::default()
            && self.oracle.validate()
            && self.pricing.validate()
            && (self.target_ratio_bps as u128) <= Perpetuals::BPS_POWER
    }
    [...]
}
```

However, it fails to call `self.funding_rate_state.validate` internally, which checks if the funding rate is less than or equal to the defined maximum threshold (`BPS_POWER`). Thus, there is no constraint on how high the funding rate may become, which is not desirable, as extremely high funding rates disproportionately benefit one side (long or short).

Remediation

Call `self.funding_rate_state.validate` within `validate`.

Patch

Fixed in e7a777f.

OS-JPT-ADV-06 [low] | Event Manipulation

Description

`ClosePositionRequestEvent` is intended to capture information about closing a position. However, a potential issue arises if sending tokens to the associated token account before the event emission, allowing manipulation of the amount field in the event, as it is directly derived from the `position_request_ata` account's amount.

events.rs RUST

```rust
#[event]
pub struct ClosePositionRequestEvent {
    pub position_request_key: Pubkey,
    pub owner: Pubkey,
    pub mint: Pubkey,
    pub amount: TokenAmount,
}
```

Hence, if sending tokens to the associated token account related to `position_request` before `ClosePositionRequest` is executed, the amount in the associated token account would increase and since the `amount` field in `ClosePositionRequestEvent` is derived from `position_request_ata` account's amount, the event would consequently reflect the manipulated amount.

Since the event should be a reliable source of information regarding the closing of a position, especially for utilization by off-chain applications, the emission of such misleading or inaccurate information may affect the operation of all applications relying on this event.

Remediation

Ensure the integrity of `ClosePositionRequestEvent`. It is advisable to validate or freeze the state of associated accounts before capturing information for the event.

Patch

Fixed in aab8cea.

05 | General Findings

Here, we present a discussion of general findings during our audit. While these findings do not present an immediate security impact, they represent anti-patterns and may result in security issues in the future.

ID	Description
OS-JPT-SUG-00	`AddPoll` lacks a length constraint on the pool name potentially colliding with pool accounts.
OS-JPT-SUG-01	The lack of multiple pool authorities and the inclusion of a predefined withdrawal delay raise potential security issues in the future.

OS-JPT-SUG-00 | Pool Name Length Constraint

Description

`AddPool` initializes a new pool, and the `params.name` field represents the pool's name and is one of the seeds that create the pool's program derived address. If the length of `params.name` exceeds 32 bytes, it may result in collisions with other program derived addresses, especially if the combination of all seeds exceeds the length limit, overlapping and interfering with different pool accounts.

add_pool.rs RUST

```rust
pub fn add_pool(ctx: Context<AddPool>, params: &AddPoolParams) -> Result<()> {
    // validate inputs
    if params.name.is_empty() || params.name.len() > 64 {
        return Err(ProgramError::InvalidArgument.into());
        }
    }
```

Remediation

Enforce a limit on the length of `params.name` to prevent collisions and ensure that the program derived address creation process remains valid. Ensuring that `params.name.len() <= 32` guarantees that the seed length remains within the constraints

Patch

Fixed in bd90f16.

OS-JPT-SUG-01 | Protocol Modifications

Description

1. In the current setup, all custody vaults share the same authority, implying that a single authority manages operations across different custody vaults. This may result in account validation issues if there are conflicts or unexpected interactions between different instances of a program that share the same authority. Moreover, if the authority becomes compromised, it may affect all the vaults.
2. There is a lack of implementation of withdrawal delay, resulting in the swift withdrawal of funds in case of a compromise or unauthorized access. Incorporating a withdrawal delay in the protocol is a protective measure against hasty and potentially malicious fund withdrawals. Additionally, it allows the protocol to detect and prevent suspicious withdrawals before they impact the broader system.
3. The protocol currently has a precision of six decimals for prices, which may limit accuracy, especially when dealing with high-value assets. Enhance precision by switching to `u128` for better representation and calculation of prices.

Remediation

1. Have a per-pool authority, which decreases the likelihood of such validation issues as each pool operates independently with its own authority.
2. Implement a withdrawal delay as a proactive security measure to enhance the protocol's safety by introducing a time buffer against potential exploits.
3. Implement the above recommendation.

A | Vulnerability Rating Scale

We rated our findings according to the following scale. Vulnerabilities have immediate security implications. Informational findings may be found in the General Findings section.

Critical	Vulnerabilities that immediately result in a loss of user funds with minimal preconditions. Examples: • Misconfigured authority or access control validation. • Improperly designed economic incentives leading to loss of funds.
High	Vulnerabilities that may result in a loss of user funds but are potentially difficult to exploit. Examples: • Loss of funds requiring specific victim interactions. • Exploitation involving high capital requirement with respect to payout.
Medium	Vulnerabilities that may result in denial of service scenarios or degraded usability. Examples: • Computational limit exhaustion through malicious input. • Forced exceptions in the normal user flow.
Low	Low probability vulnerabilities, which are still exploitable but require extenuating circumstances or undue risk. Examples: • Oracle manipulation with large capital requirements and multiple transactions.
Informational	Best practices to mitigate future security risks. These are classified as general findings. Examples: • Explicit assertion of critical internal invariants. • Improved input validation.

B | Procedure

As part of our standard auditing procedure, we split our analysis into two main sections: design and implementation.

When auditing the design of a program, we aim to ensure that the overall economic architecture is sound in the context of an on-chain program. In other words, there is no way to steal funds or deny service, ignoring any chain-specific quirks. This usually requires a deep understanding of the program's internal interactions, potential game theory implications, and general on-chain execution primitives.

One example of a design vulnerability would be an on-chain oracle that could be manipulated by flash loans or large deposits. Such a design would generally be unsound regardless of which chain the oracle is deployed on.

On the other hand, auditing the program's implementation requires a deep understanding of the chain's execution model. While this varies from chain to chain, some common implementation vulnerabilities include reentrancy, account ownership issues, arithmetic overflows, and rounding bugs.

As a general rule of thumb, implementation vulnerabilities tend to be more "checklist" style. In contrast, design vulnerabilities require a strong understanding of the underlying system and the various interactions: both with the user and cross-program.

As we approach any new target, we strive to comprehensively understand the program first. In our audits, we always approach targets with a team of auditors. This allows us to share thoughts and collaborate, picking up on details that the other missed.

While sometimes the line between design and implementation can be blurry, we hope this gives some insight into our auditing procedure and thought process.