Port Sundial

Audit

Presented by:

OtterSec contact@osec.io

David Chen ra@osec.io

Robert Chen notdeghost@osec.io

Parth Shastri cppio@osec.io

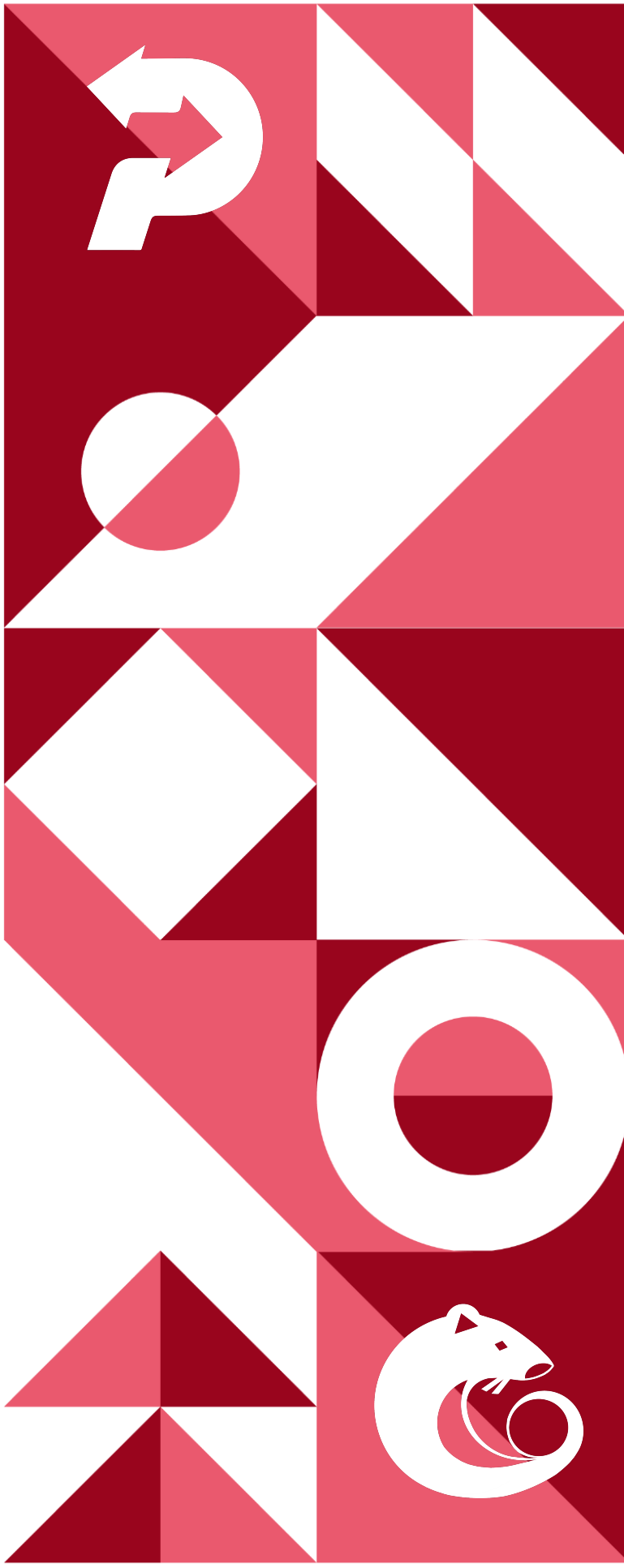


Table of Contents

01 | Executive Summary

Overview

Port Finance engaged OtterSec to perform an assessment of their `sundial` program.

This assessment was conducted between March 14th and April 1st, 2022, with a focus on the following:

- Provide meaningful recommendations to reduce program attack surface
- Evaluate integrity of the Sundial program at an implementation and design level

Critical vulnerabilities were communicated to the team prior to the delivery of the report to speed up remediation.

We delivered final confirmation of the patches April 4th, 2022.

Key Findings

The following is a summary of the major findings in this audit.

- 7 findings total
- 3 vulnerabilities which could lead to loss of funds
    - OS-PRT-ADV-00: Improper liquidation rounding liquidates excess collateral
    - OS-PRT-ADV-01: Inaccurate yield token calculations
    - OS-PRT-ADV-02: Configuration change zeroes critical fields

As part of this audit, we also provided proof of concepts for each vulnerability to easily prove exploitability and enable simple regression testing. These scripts can be found at https://osec.io/pocs/port-sundial. For a full list, see Appendix C.

We also observed the following:

- The team was very helpful and responsive to our feedback
- Code quality was extremely high and most issues we identified were relatively subtle
- Overall economic design of the Sundial program was sound

02 | Scope

We received the program from Port Finance and began the audit March 14th, 2022. The source code was delivered to us in a public git repository at https://github.com/port-finance/sundial. This audit was performed against commit c0879ae.

We were asked to audit the Sundial smart contract program, located in the GitHub repository at `programs/sundial`.

A full list of program files and hashes can be found in Appendix A. The Sundial program contains two main sets of instructions.

Name	Description
Borrowing	Manages user profiles, collateral, and loans
Lending	Sundial principal and yield token management

These are found in the `borrowing_instructions` and `lending_instructions` folders respectively.

While we did not receive a prior or internal audit, we consulted the litepaper for Sundial which helped explain the codebase and economic design.

03 | 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 onchain program. In other words, there is no way to steal tokens or deny service, ignoring any Solana specific quirks such as account ownership issues. An example of a design vulnerability would be an onchain oracle which could be manipulated by flash loans or large deposits.

On the other hand, auditing the implementation of the program requires a deep understanding of Solana's execution model. Some common implementation vulnerabilities include account ownership issues, arithmetic overflows, and rounding bugs. For a non-exhaustive list of security issues we check for, see Appendix B. While Sundial's code was overall very solid, we identified a rounding issue here which would allow an attacker to create under-collateralized loans, ultimately stealing funds from the protocol (OS-PRT-ADV-00).

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. Here we identified a misallocation of yield tokens, letting a user who deposited late steal interest payments from other users (OS-PRT-ADV-01).

As we approach any new target, we strive to get a comprehensive understanding of the program first. In our audits, we always approach any target in a team of at least two. This allows us to share thoughts and collaborate, picking up on details that the other might have missed.

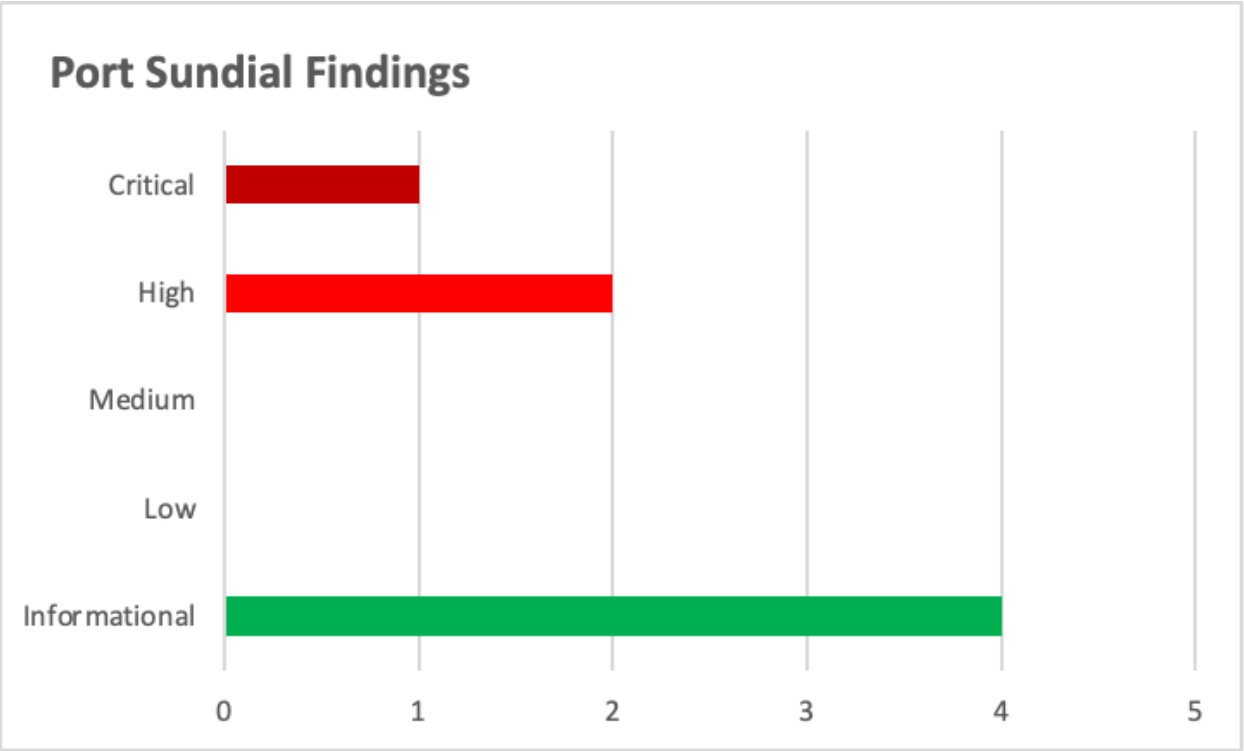
While sometimes the line between design and implementation can be blurry, we hope this gives some insight into our auditing procedure and thought process.

04 | Findings

Overall, we report 7 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 don't have an immediate impact but will help mitigate future vulnerabilities.

The below chart displays the findings by severity.



Proof of Concepts

For each vulnerability, we created a proof of concept to prove exploitability and enable easy regression testing. We recommend integrating these as part of a comprehensive test suite. The proof of concept directory structure can be found in Appendix C.

A GitHub repository containing these proof of concepts can be found at https://osec.io/pocs/port-sundial.

To run a POC:

```
./run.sh <directory name>
```

For example,

```
./run.sh os-prt-adv-00
```

The relevant code for each POC can be found in `<dir>/*.ts`. These POCs make use of Sundial's existing test framework.

05 | Vulnerabilities

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

Rating criterion can be found in Appendix D.

ID	Severity	Status	Description
OS-PRT-ADV-00	**Critical**	**Resolved**	Improper liquidation rounding allows the creation of an attacker to under-collateralize an obligation, leading to loss of funds
OS-PRT-ADV-01	**High**	**Resolved**	Incorrect yield token calculations allow attackers who deposit late to steal interest payments from earlier users.
OS-PRT-ADV-02	**High**	**Resolved**	Missing field assignment in admin instruction causes price calculations to be wildly incorrect, leading to loss of funds.

OS-PRT-ADV-00 [Critical] [Resolved]: Excess Collateral Liquidations

Description

The program instruction `liquidate_sundial_profile` rounds up the amount of collateral received by the liquidator. This results in the liquidators being able to create an undercollateralized account by repaying much less than the value of the collateral received repeatedly.

instructions/borrow_instructions/liquidate_sundial_profile.rs

```
let withdraw_value = log_then_prop_err!(collateral_to_withdraw
    .config
    .liquidation_config
    .get_liquidation_value(loan_to_repay.asset.get_value(repay_amount)?));
let withdraw_amount = log_then_prop_err!(collateral_to_withdraw
    .asset
    .get_amount(withdraw_value)
    .and_then(|d| d.try_ceil_u64()));
```

Notice how the `withdraw_amount` is ceilinged in the calculation. The minimum amount received by the liquidator is one collateral token as long as the repay token has a non zero `liquidation_value`.

If the value of the repaid token is lower than the collateral token, a liquidator can then repeatedly repay with a low valued repay token and receive a much higher valued collateral token.

While each repayment only causes a small discrepancy between the collateral and obligation amount, eventually an attacker will be able to push the value of the collateral lower than the obligation. Note that each of these operations is still profitable to the attacker, similar to the previous spl-token-lending rounding bug.

Because the attacker is profiting from such a transaction, the lending protocol must be losing money. This leads to a loss of funds scenario for the Sundial program.

Proof of Concept

More concretely, consider the following scenario:

1. Attacker deposits some token A, which has a high value per minimum token unit (BTC for example)
2. Attacker borrows some token B, which has a low value per minimum token unit (SOL for example)
3. The price of SOL goes up, which makes the attacker liquidatable
4. The attacker liquidates themselves, repaying a single lamport and receiving 1 satoshi.
5. Because such a liquidation lowers the health of the account, the user is able to do this repeatedly, and the account remains liquidatable throughout.
6. The loan is never repaid fully and the collateral is drained by the liquidator. The lending protocol ends up with an undercollateralized account.
7. The attacker keeps both the collateral and the borrowed asset, in essence stealing the obligation from the lending protocol

We constructed a proof-of-concept which creates an undercollateralized account by repeatedly repaying a low-value token while receiving a high value collateral token.

In our proof-of-concept for demonstration purposes, we used two fake tokens with a large value difference. A real world example of this with less extreme value differentials could be found between USDC and BTC.

After a series of malicious liquidation operations, we are able to entirely drain the lower value collateral, leaving behind a severely undercollateralized account.

```
total collateral value before: 100000283919052573356860
total loan value before:       902000000000000000000
total collateral value after:  100000000000000000
total loan value:              901800000000000000000
```

This leads to a loss of funds scenario for the lending protocol. If an attacker maliciously creates an undercollateralized account, they could simply keep the loan, never repaying the obligation. Because the value of the loan is higher than the collateral, the lending protocol would be forced to make up the difference.

Remediation

The lending protocol should ensure that in the general case, the health of the account increases. A liquidation should generally never result in bringing the user closer to becoming undercollateralized.

At the same time, it will be necessary to ensure that small obligations can be properly liquidated in a profitable manner for the liquidator, even if that liquidation might decrease the health of the account.

This could be done similar to spl-token-lending's `reserve.calculate_liquidation` function which considers such small amounts as an edge case.

```
// Close out obligations that are too small to liquidate normally
if liquidity.borrowed_amount_wads < LIQUIDATION_CLOSE_AMOUNT.into() {
    // settle_amount is fixed, calculate withdraw_amount and repay_amount
    settle_amount = liquidity.borrowed_amount_wads;

    let liquidation_value = liquidity.market_value.try_mul(bonus_rate)?;
    match liquidation_value.cmp(&collateral.market_value) {
        Ordering::Greater => {
```

Patch

Ensure risk factor decreases during liquidation, fixed in #77.

OS-PRT-ADV-01 [High] [Resolved]: Inaccurate Yield Token Calculations

Description

When depositing liquidity and minting principal and yield tokens, the user is given one yield token for each underlying liquidity token they deposit. This calculation is incorrect. The user should instead receive one yield token for each principal token.

When calculating tokens returned to the user, the principal token amount is properly backdated to the start of Sundial. The intent of this calculation is to ensure that if users deposit tokens late, they receive the same amount of principal tokens as if they had deposited at the start of the Sundial instance.

instructions/lending_instructions/deposit_and_mint_tokens.rs

```
// We calculate how much liquidity is deposited if we deposit it at
the very beginning of [Sundial].
let principal_token_amount =
start_exchange_rate.collateral_to_liquidity(unwrap_int!(ctx
    .accounts
    .sundial_port_lp_wallet
    .amount
    .checked_sub(existed_lp_amount)))?;
```

However, this calculation only applies to the minted principal tokens. As seen below, the amount of deposited liquidity is used directly when determining how many yield tokens to mint.

instructions/lending_instructions/deposit_and_mint_tokens.rs

```
pub fn process_deposit_and_mint_tokens(
    ctx: Context<DepositAndMintTokens>,
    amount: u64,
) -> ProgramResult {
```

instructions/lending_instructions/deposit_and_mint_tokens.rs

```
log_then_prop_err!(mint_to(
    create_mint_to_cpi(
```

```
        ctx.accounts.yield_token_mint.to_account_info(),
        ctx.accounts.user_yield_token_wallet.to_account_info(),
        ctx.accounts.sundial_authority.to_account_info(),
        seeds!(ctx, sundial, authority),
        ctx.accounts.token_program.to_account_info(),
    ),
    amount
));
```

This relatively subtle bug means that if a user deposits near the end of the period, they will receive more yield tokens then they deserve. A malicious user who deposited a very large amount could take an arbitrarily large proportion of the yield.

For example, a user who deposits 100 tokens 1 slot before the end of the period might receive 80 principal tokens and 100 yield tokens. Because the yield token amount is proportionally larger, the user who deposited late takes some yield that would have otherwise gone to the other users, despite providing liquidity for only a single slot.

Another way to think about this is: the value of the returned principal tokens and yield tokens must be equal to the deposited liquidity.

This creates a misaligned economic structure where users are incentivized to delay adding tokens to the Sundial program until the very end of the period.

Proof of Concept

More concretely, consider the following scenario

1. Victim deposits liquidity tokens at the beginning of a Sundial
2. Attacker waits until the slot before the Sundial ends and then deposits liquidity tokens. Note that this does not have to be timed perfectly. Doing the deposit anytime close to the end of the Sundial period is enough to be profitable for the attacker.
3. Attacker waits for the next slot and redeems the principle and yield tokens for liquidity tokens, making profit

We modeled the potential victim and attacker balances with the following set of equations:

Victim Principle Redeem Amount: $L_1(1-F)$

Victim Yield Redeem Amount: $\left(L_1 + \frac{L_2}{R}\right)(R-1)\frac{L_1}{L_1+L_2}$

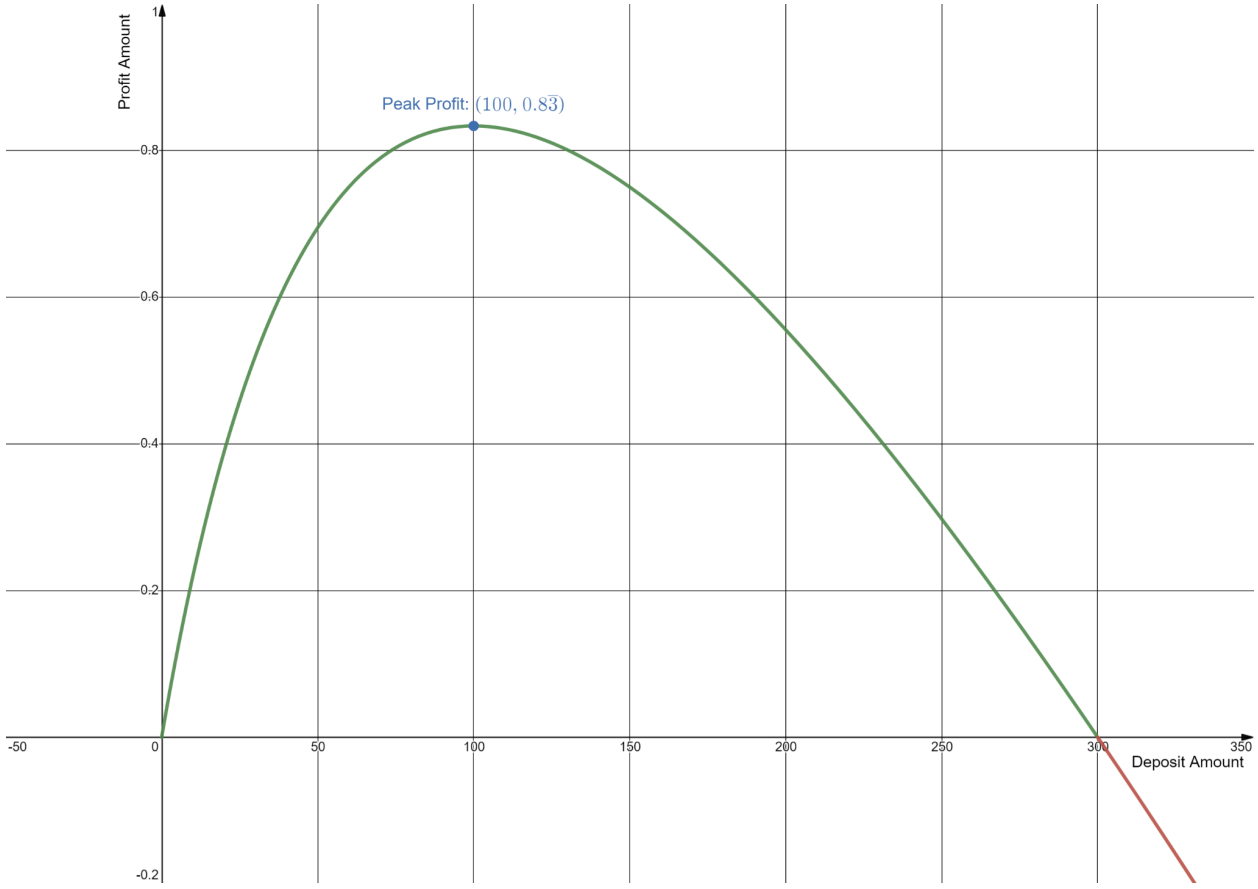
Attacker Principle Redeem Amount: $\frac{L_2}{R}(1-F)$

Attacker Yield Redeem Amount: $\left(L_1+\frac{L_2}{R}\right)(R-1)\frac{L_2}{L_1+L_2}$

Where

L_1 is the liquidity deposited by the victim at the start of the Sundial,
L_2 is the liquidity deposited by the attacker right before the Sundial ends,
R is the increase in value of the collateral tokens over the course of the Sundial,
F is the lending fee.

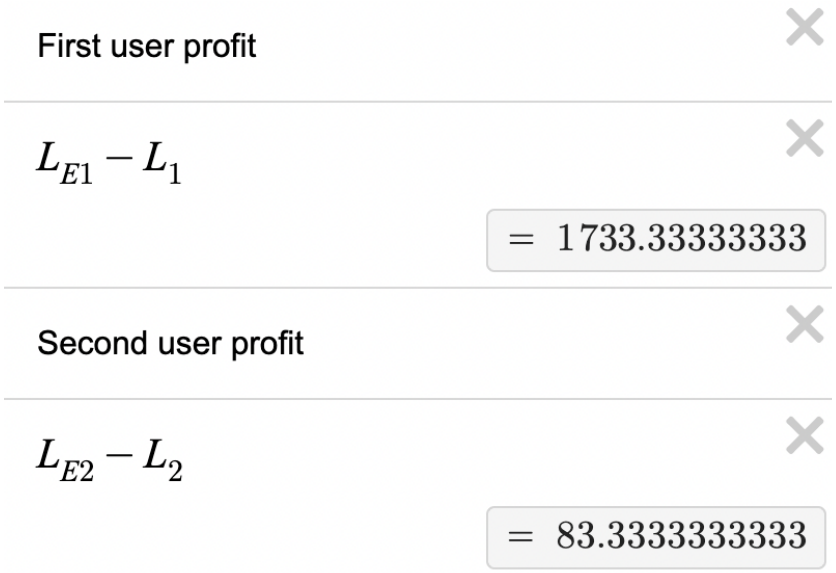
With these equations, we produced the following graph for the net profit of the attacker as a function of the amount of liquidity they deposited, assuming a yield of 20% and a fee of 1% ($R = 1.2$, $F = 0.01$). The units are in the percent of the victim's deposit amount, demonstrating that this attack is indeed profitable despite the fee.



For instance, if the victim deposited 1000 USDC at the start of the Sundial, then the attacker could deposit 1000 USDC at the end and have a net profit of 8.33 USDC.

A complete Desmos instance with our math can be found at https://www.desmos.com/calculator/rdcrft1uvd.

Note that this calculator does not include a graph. Instead, refer to the side panel.



Remediation

The program should mint the same amount of yield tokens as principal tokens. This results in less yield tokens being given to the attacker, resulting in no profit.

Patch

Use principal token amount as the amount of yield tokens minted, fixed in #78.

OS-PRT-ADV-02 [High] [Resolved]: Config Changes Zero Critical Fields

Description

The privileged `ChangeSundialConfig` and `ChangeSundialCollateralConfig` instructions clobber the `liquidity_decimals` and `collateral_decimals` fields respectively.

When initializing the Sundial instance, note how the `liquidity_decimals` field is assigned after the configuration assignment.

instructions/lending_instructions/initialize_sundial.rs

```
sundial.config = config.into();
sundial.sundial_market = ctx.accounts.sundial_market.key();
sundial.oracle = oracle;
sundial.config.liquidity_decimals = ctx.accounts.port_liquidity_mint.decimals;

emit!(InitializeSundialEvent {
    sundial: sundial.key(),
    duration_in_seconds,
});

Ok(())
```

This is because the implementation for `From<SundialInitConfigParams>` will silently zero all other fields such as `liquidity_decimals`.

instructions/lending_instructions/initialize_sundial.rs

```
impl From<SundialInitConfigParams> for SundialConfig {
    fn from(config: SundialInitConfigParams) -> Self {
        SundialConfig {
            lending_fee: Fee {
                bips: config.lending_fee,
            },
            borrow_fee: Fee {
                bips: config.borrow_fee,
            },
            liquidity_cap: LiquidityCap {
                lamports: config.liquidity_cap,
```

```
            },
            ..SundialConfig::default()
        }
    }
}
```

However, the `liquidity_decimals` field is not reassigned to with the `change_sundial_config` handler. This means that when an admin tries to change the sundial configuration, it will silently zero this amount.

instructions/lending_instructions/change_sundial_config.rs

```
ctx.accounts.sundial.config = config.into();
emit!(ChangeSundialConfigEvent {
    sundial: ctx.accounts.sundial.key(),
    config: ctx.accounts.sundial.config.clone(),
});
Ok(())
```

This field is used when initializing obligations. By zeroing this field, this has the effect of multiplying the value of all new obligations by a large constant factor, leading to a wildly inaccurate price on subsequent transactions.

instructions/borrowing_instructions/mint_sundial_liquidity_with_collateral.rs

```
SundialProfileLoan::init_loan(
    amount,
    oracle_info,
    ctx.accounts.sundial.key(),
    &ctx.accounts.clock,
    ctx.accounts.sundial.end_unix_time_stamp,
    ctx.accounts.sundial.config.liquidity_decimals,
)
```

A very similar issue exists with changing sundial collateral configuration.

instructions/borrowing_instructions/change_sundial_collateral_config.rs

```
ctx.accounts.sundial_collateral.sundial_collateral_config = config.into();
```

```
Ok(())
```

This is especially dangerous, because changing the collateral configuration would massively overvalue the user's collateral, allowing users to borrow endlessly from the Sundial program, draining the pool.

Proof of Concept

Consider the following scenario

1. A user deposits collateral
2. An admin runs the `change_sundial_collateral_config` instruction. This zeroes the collateral decimals field, massively overvaluing any future collateral deposits.
3. The user mints principle tokens for far more than the market value of the collateral.
4. The user redeems the principle tokens after the Sundial ends, making a massive amount of profit.

Remediation

The program should keep the `liquidity_decimals` and `collateral_decimals` fields the same when the config is changed.

Patch

Properly propagate config changes, fixed in #78.

06 | General Findings

Here we present a discussion of general findings during our audit. While these findings do not present an immediate security impact, they do represent antipatterns and could introduce a vulnerability in the future.

ID	Description
OS-PRT-SUG-00	Possible seed collision between Sundial and Sundial collateral
OS-PRT-SUG-01	Sundial configuration From impl
OS-PRT-SUG-02	Replace Anchor macros
OS-PRT-SUG-03	Clippy owned string instance

OS-PRT-SUG-00 [Resolved]: Potential Seed Collision

Description

Both the addresses of `Sundial` and `SundialCollateral` are derived from the `SundialMarket` address, but the `InitializeSundial` instruction does not use a fixed suffix.

instructions/lending_instructions/initialize_sundial.rs

```
#[account(
    init,
    payer = owner,
    seeds = [
        sundial_market.key().as_ref(),
        name.as_ref()
    ],
    bump = pda_bump
)]
pub sundial: Account<'info, Sundial>,
```

instructions/borrowing_instructions/initialize_sundial_collateral.rs

```
#[account(
    init,
    payer = owner,
    seeds = [
        sundial_market.key().as_ref(),
        name.as_ref(),
        b"collateral"
    ],
    bump = pda_bump
)]
pub sundial_collateral: Account<'info, SundialCollateral>,
```

This results in the owner being able to create a `Sundial` and a `SundialCollateral` at the same address by appending `b"sundial"` to the collateral name and passing it to `InitializeSundial`.

For example, a Sundial instance created with “Xcollateral” would collide with a SundialCollateral instance created with “X”.

This could potentially lead to a usability concern if the seeds could be influenced by a malicious user. However, due to the low risk and impact, we rated this as informational as opposed to a denial of service concern.

Remediation

The `seeds` for `InitializeSundial` should include a suffix such as `b"sundial"` to prevent collisions.

Patch

Sundial initialization uses `b”sundial”` as a suffix.

instructions/lending_instructions/initialize_sundial.rs

```
          payer = owner,
          seeds = [
              sundial_market.key().as_ref(),
-             name.as_ref()
+             name.as_ref(),
+             b"sundial"
          ],
          bump = pda_bump
```

OS-PRT-SUG-01 [Resolved]: Sundial Config From Impl

Description

The `From<SundialInitConfigParams>` and `From<SundialCollateralConfigParams>` definitions use a security antipattern by implicitly zeroing extraneous fields.

For example, consider the initialization for `SundialConfig`.

src/instructions/lending_instructions/initialize_sundial.rs

```
impl From<SundialInitConfigParams> for SundialConfig {
    fn from(config: SundialInitConfigParams) -> Self {
        SundialConfig {
            lending_fee: Fee {
                bips: config.lending_fee,
            },
            borrow_fee: Fee {
                bips: config.borrow_fee,
            },
            liquidity_cap: LiquidityCap {
                lamports: config.liquidity_cap,
            },
            ..SundialConfig::default()
        }
    }
}
```

In particular, the use of `..SundialConfig::default()` means that any additional parameters will silently be zeroed when assigned to from `SundialInitConfigParams`.

A similar issue exists with `SundialCollateralConfigParams`.

instructions/borrowing_instructions/initialize_sundial_collateral.rs

```
impl From<SundialCollateralConfigParams> for SundialCollateralConfig
{
    fn from(config: SundialCollateralConfigParams) -> Self {
```

```
        SundialCollateralConfig {
            ltv: LTV { ltv: config.ltv },
            liquidation_config: LiquidationConfig {
                liquidation_threshold: config.liquidation_threshold,
                liquidation_penalty: config.liquidation_penalty,
            },
            liquidity_cap: LiquidityCap {
                lamports: config.liquidity_cap,
            },
            ..SundialCollateralConfig::default()
        }
    }
}
```

Remediation

We recommend removing the use of `..Sundial*Config::default()` and explicitly assigning values to each field, for example a sane default or the original configuration value. This will ensure that future changes do not silently break.

Patch

Removed the use of `..Sundial*Config::default()`.

instructions/lending_instructions/initialize_sundial.rs

```
-            ..SundialConfig::default()
+            liquidity_decimals: 0,
+            _config_padding: [0; 5],
```

OS-PRT-SUG-02: Replace Macros With Anchor Constraints

Description

The use of macros in this codebase makes it difficult to efficiently confirm code security.

For example, `check_sundial_owner` is defined separately in `sundial-derives`, and ensures that the transaction is signed by the owner of the market.

sundial-derives/src/lib.rs

```
#[proc_macro_derive(CheckSundialMarketOwner)]
pub fn check_sundial_market_owner(input: TokenStream) -> TokenStream
{
    let ast = parse_macro_input!(input as DeriveInput);
    let name = &ast.ident;
    (quote! {
        impl<'a> crate::helpers::CheckSundialMarketOwner for
#name<'a> {
            fn check_sundial_market_owner(&self) -> ProgramResult {
                vipers::assert_keys_eq!(
                    self.sundial_market.owner,
                    *self.owner.key,
                    SundialError::InvalidOwner,
                    "Invalid Sundial Market Owner"
                );
                Ok(())
            }
        }
    })
    .into()
}
```

This is functionally equivalent to Anchor's `has_one` constraint [0].

instructions/lending_instructions/initialize_sundial.rs

```
#[account(mut)]
pub owner: Signer<'info>,
// [0] #[account(has_one = owner)]
```

```
pub sundial_market: Box<Account<'info, SundialMarket>>,
```

Anchor checks are inline and makes it easier to confirm the security of the code without looking at the definition for such macros. In addition, it's a more standardized way to check for such conditions, avoiding possible errors in the macro implementation.

Remediation

We would recommend replacing the following macros with the corresponding Anchor implementation:

- CheckSundialMarketOwner
- CheckSundialProfileMarket
- CheckSundialOwner

OS-PRT-SUG-03 [Resolved]: Clippy Owned Instance

Description

`cargo clippy` should be run prior to deployment.

```
warning: this creates an owned instance just for comparison
   --> sundial-derives/src/lib.rs:147:24
    |
147 |                     if ident.to_string() == field_name {
    |                        ^^^^^^^^^^^^^^^^ help: try: `ident`
    |
    = note: `#[warn(clippy::cmp_owned)]` on by default
    = help: for further information visit
https://rust-lang.github.io/rust-clippy/master/index.html#cmp_owned

warning: `sundial-derives` (lib) generated 1 warning
warning: `sundial-derives` (lib) generated 1 warning (1 duplicate)
```

While this particular instance is in a derive and thus only results in increased compilation time, this is a best practice and can catch other vulnerabilities or security antipatterns.

Remediation

Implement the suggestions from `cargo clippy`.

In this case, remove the unnecessary `.to_string()`.

Patch

Removed the unnecessary `.to_string()`.

07 | Appendix

Appendix A: Program Files

Below are the files in scope for this audit and their corresponding truncated sha256 hashes.

```
Cargo.toml                                        393aa2fba3975f0bdd95371c8cdb9b0f
Xargo.toml                                        815f2dfb6197712a703a8e1f75b03c69
src
  error.rs                                        b16db71ecbaad3f28ac0e339c8f293a1
  helpers.rs                                      6d36f9c5ebc937cbaaf14814592cb1c9
  lib.rs                                          4a890034508f4d74bbb51078ee24955d
  state.rs                                        404063fd4dd1fb22a13ad3295ddaa4a6
  instructions
    mod.rs                                        318fc183a9efb13caf2791a16a780959
    borrowing_instructions
      change_sundial_collateral_config.rs         cb8da2fe1d115d8a75898ff9515ef86a
      deposit_sundial_collateral.rs               818aad15b88db1e4eeeebdda2b4d00b7
      initialize_sundial_collateral.rs            0571dfd5eb2ace6e17c6a7b013c8cb8d
      initialize_sundial_profile.rs               f96ebb24c9651c396c5934c9c31cb849
      liquidate_sundial_profile.rs                8eae7ea0aeeeb2ac4263a8c51ab17431
      mint_sundial_liquidity_with_collateral.rs   49aceb8420bd6a2865e5eecbcb45c417
      mod.rs                                      2e6f7a438e92b21e8613984054bd401d
      refresh_sundial_collateral.rs               fc4b5592f593745a07cf1e6ad5395347
      refresh_sundial_profile.rs                  fae896f179b42647632a1a5b6c7ae165
      repay_sundial_liquidity.rs                  510ba235d37e84d062de6dfe9cd92179
      withdraw_sundial_collateral.rs              1da4d0d6fb260d8c9e1c034132a23759
    lending_instructions
      change_sundial_config.rs                    5b00789f9933d563a0de9ffef36ab5b6
      deposit_and_mint_tokens.rs                  60b22ac44150f8bb5bb9adde602081ee
      initialize_sundial.rs                       a97fa6f378dbef208e1651b8dd339cc4
      initialize_sundial_market.rs                0c9d0405e48a51070e2ddc1a492e0ddc
      mod.rs                                      8dcb9e4ac66fc049d0496b662468f3a0
      redeem_lp.rs                                6a26e4835fa838d5482f54455995087b
      redeem_principle_token.rs                   d668fb22240df6e7c107dfa3b93b9ed0
      redeem_yield_token.rs                       a03355b89ee6efba0923c50c1c4810c8
```

Appendix B: Implementation Security Checklist

Unsafe arithmetic

Integer underflows or overflows	Unconstrained input sizes could lead to integer over or underflows, causing potentially unexpected behavior. Ensure that for unchecked arithmetic, all integers are properly bounded.
Rounding	Rounding should always be done against the user to avoid potentially exploitable off-by-one vulnerabilities.
Conversions	Rust `as` conversions can cause truncation if the source value does not fit into the destination type. While this is not undefined behavior, such truncation could still lead to unexpected behavior by the program.

Account security

Account Ownership	Account ownership should be properly checked to avoid type confusion attacks. For Anchor, the safety of unchecked accounts should be clearly justified and immediately obvious.
Accounts	For non-Anchor programs, the type of the account should be explicitly validated to avoid type confusion attacks.
Signer Checks	Privileged operations should ensure that the operation is signed by the correct accounts.
PDA Seeds	PDA seeds are uniquely chosen to differentiate between different object classes, avoiding collision.

Input Validation

Timestamps	Timestamp inputs should be properly validated against the current clock time. Timestamps which are meant to be in the future should be explicitly validated so.
Numbers	Sane limits should be put on numerical input data to mitigate the risk of unexpected over and underflows. Input data should be constrained to the smallest size type possible, and upcasted for unchecked arithmetic.
Strings	Strings should have sane size restrictions to prevent denial of service conditions

Internal State	If there is internal state, ensure that there is explicit validation on the input account's state before engaging in any state transitions. For example, only open accounts should be eligible for closing.

Miscellaneous

Libraries	Out of date libraries should not include any publicly disclosed vulnerabilities
Clippy	`cargo clippy` is an effective linter to detect potential anti-practices.
Signer Authority	Ensure that cross program invocations with the program's signer authority are properly permissioned.

Appendix C: Proof of Concepts

Below are the provided proof of concept files and their truncated sha256 hashes.

```
os-prt-adv-00
  clone.sh                                  e465d5cbde4b0d3dae64eccdd8bc5df5
  sundial_collateral.spec.ts                4603f0a0247d71d2d9a0d2f87c0d17dd
os-prt-adv-01
  clone.sh                                  e465d5cbde4b0d3dae64eccdd8bc5df5
  incorrect_yield.spec.ts                   fb9be8a1f5eba1f83ed6a52c1c7fbf72
os-prt-adv-02
  clone.sh                                  e465d5cbde4b0d3dae64eccdd8bc5df5
  incorrect_config_decimals.spec.ts         b7012049b654e21f91014817e7dd581d
```

Appendix D: Vulnerability Rating Scale

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

Critical	Vulnerabilities which **immediately** lead to loss of user funds with minimal preconditions Examples: - Misconfigured authority/token account validation - Rounding errors on token transfers
High	Vulnerabilities which **could** lead to 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 which could lead to denial of service scenarios or degraded usability. Examples: - Malicious input cause computation limit exhaustion - Forced exceptions preventing normal use
Low	Low probability vulnerabilities which could still be 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 - Uncaught Rust errors (vector out of bounds indexing)