Cega

Audit

Presented by:

OtterSec contact@osec.io

Robert Chen notdeghost@osec.io

Rajvardhan Agarwal raj@osec.io

Aleksandre Khokhiashvili khokho@osec.io

Table of Contents

01 | Executive Summary

Overview

Cega Finance engaged OtterSec to perform an assessment of the `cega-vault` program prior to deployment.

This assessment was conducted between May 16th and June 3rd, 2022.

Critical vulnerabilities were communicated to the team prior to the delivery of the report to speed up remediation. After delivering our audit report, we worked closely with the team to streamline patches and confirm remediation.

Key Findings

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

- 13 findings total
- 6 vulnerabilities which could lead to loss of funds
    - OS-CEG-ADV-00: Denial of withdrawal due to insufficient checks on vault
    - OS-CEG-ADV-01: Lower payoff for users due to missing account reload
    - OS-CEG-ADV-02: Max payout in a knock-in event due to missing length check
    - OS-CEG-ADV-03: Max payout in a knock-in event due to insufficient checks
    - OS-CEG-ADV-04: Denial of withdrawal due to missing length check
    - OS-CEG-ADV-05: Vault denial of service due to incorrect calculation

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

We also observed the following:

- Code quality of the program was high and overall design was solid
- The use of Anchor prevented many implementation-level bugs
- Team was very knowledgeable and responsive in remediating vulnerabilities

02 | Scope

We received the program from Cega Finance and began the audit on May 16th, 2022. The source code was delivered to us at https://github.com/cega-fi/cega-vault. This audit was performed against commit 3ad8f80.

A brief description of the programs is as follows. A full list of program files and hashes can be found in Appendix A.

Name	**Description**
cega-vault	The program functionality includes: - Depositing USDC into vaults to receive redeemable tokens - Redeeming USDC from a vault

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. One such issue that was identified in the `cega-vault` program was an account validation issue that would allow an attacker to lock-in USDC tokens (OS-CEG-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 an issue that would allow an attacker to block withdrawal for other users (OS-CEG-ADV-04).

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 two. This allows us to share thoughts and collaborate, picking up on details that the other missed.

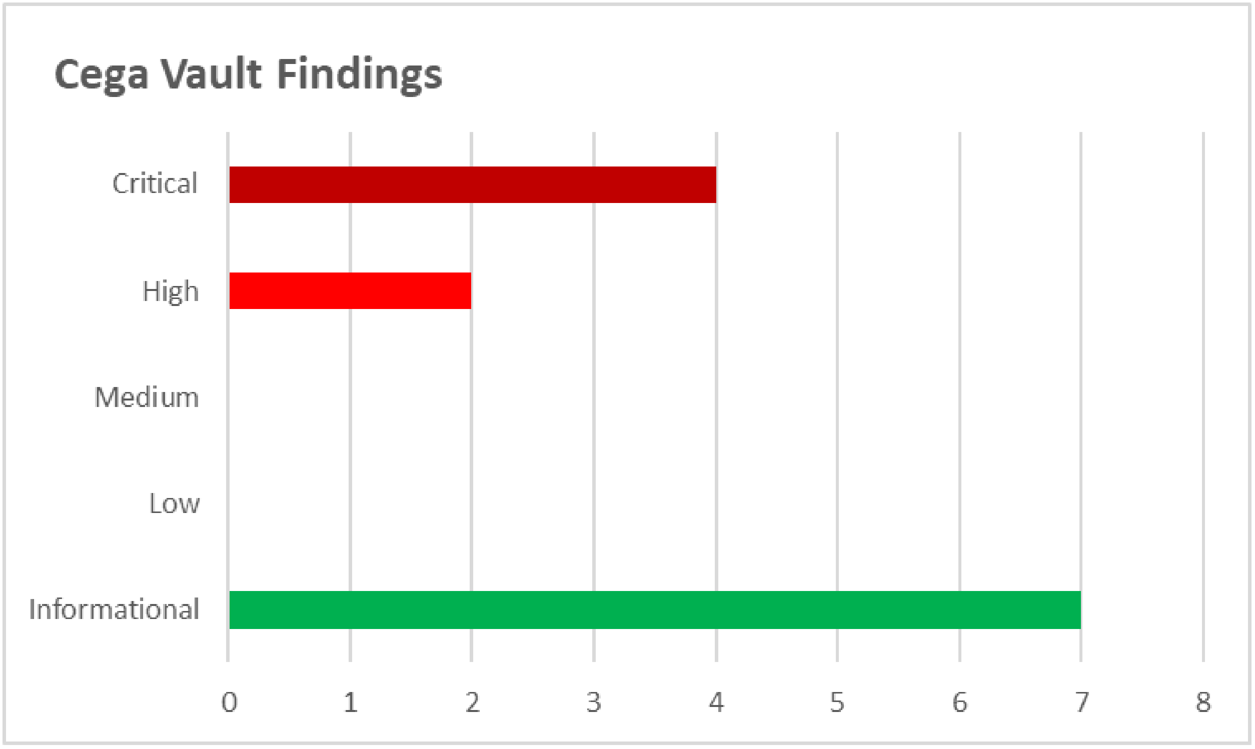
While sometimes the line between design and implementation can be blurry, we hope this gives some insight into our auditing procedure and thought process.

04 | Findings

Overall, we report 13 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.



Proofs of Concept

For each vulnerability we created a proof of concept to 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.

These proofs of concept can be found at https://osec.io/pocs/cega-vault.

To run a POC:

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

For example,

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

Each proof of concept comes with its own patch file which modifies the existing test framework to demonstrate the relevant vulnerability. We also recommend integrating these patches into the test suite to prevent regressions.

05 | 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 criterion can be found in Appendix D.

ID	Severity	Status	Description
OS-CEG-ADV-00	**Critical**	**Resolved**	Denial of withdrawal due to insufficient checks in DepositVault.
OS-CEG-ADV-01	**Critical**	**Resolved**	Not reloading the mint account after cross-program invocation leads to incorrect calculations for withdrawn funds.
OS-CEG-ADV-02	**Critical**	**Resolved**	Attackers can claim full payout in case of a knock-in event due to missing length checks on remaining accounts.
OS-CEG-ADV-03	**Critical**	**Resolved**	Attackers can claim full payout in case of a knock-in event due to insufficient checks on OptionBarrier and oracle accounts.
OS-CEG-ADV-04	**High**	**Resolved**	An attacker has the ability to block withdrawals of other users.
OS-CEG-ADV-05	**High**	**Resolved**	Integer subtraction overflow leads to denial of service for all vaults associated with a product.

OS-CEG-ADV-00 [Critical] [Resolved]: Denial of withdrawal

Description

The `deposit_vault` instruction does not check if the provided vault account is associated with the product account. An attacker can exploit this vulnerability to transfer USDC tokens to a `TokenAccount` associated with a different product, while still increasing the underlying amount for the vault [0].

This could block other users from withdrawing funds, as the product associated `TokenAccount` might not have enough liquidity. The following snippets show the affected code.

cega-vault/src/context.rs

```
#[derive(Accounts)]
#[instruction(args: DepositVaultArgs)]
pub struct DepositVault<'info> {
[...]
    #[account(
        mut,
        seeds = [product.product_name.as_ref().strip()],
        bump = product.product_nonce,
        constraint = product_underlying_token_account.mint ==
underlying_mint.key() @ VaultError::InvalidUnderlyingMint
    )]
    pub product: Box<Account<'info, Product>>,

    #[account(
        mut,
        seeds = [vault.product_name.as_ref().strip(),
&vault.vault_number.to_le_bytes()],
        bump = vault.vault_nonce,
    )]
    pub vault: Box<Account<'info, Vault>>,
[...]
    #[account(
        mut,
        seeds = [PRODUCT_UNDERLYING_SEED.as_bytes(),
product.key().as_ref()],
        bump = product.product_underlying_token_account_nonce
    )]
    pub product_underlying_token_account: Box<Account<'info,
TokenAccount>>,
```

cega-vault/src/lib.rs

```
            // Add to vault's total deposit
            let vault = &mut ctx.accounts.vault;
            vault.underlying_amount = vault
                .underlying_amount
                .checked_add(user_deposit_amount)
                .unwrap(); // [0]

            // Add to product's total deposit
            let product = &mut ctx.accounts.product;
            product.underlying_amount = product
                .underlying_amount
                .checked_add(user_deposit_amount)
                .unwrap();
```

Proof of Concept

1. Create two product accounts and vaults associated with each product.
2. The user deposits USDC into the vault associated with the first product and receives redemption tokens.
3. The attacker deposits USDC into the vault associated with the first product but uses the second product account instead. The attacker also receives redemption tokens.
4. Admin processes the trade and the yield generation period begins.
5. During the yield period, the attacker places a withdrawal request on the queue.
6. The yield generation period ends. The protocol calculates the vault payoff and collects the fees.
7. The attacker's withdrawal request gets processed and they get paid out the full principal amount and tokens as per the APR.
8. The user requests a withdrawal in the next yield period, but the withdrawal fails as the original product associated `TokenAccount` does not have enough USDC tokens.

Remediation

Derive the vault seed with `product.product_name` instead of `vault.product_name`.

Patch

Vault seed is now derived using `product.product_name`. Fixed in #156.

OS-CEG-ADV-01 [Critical] [Resolved]: Incorrect withdrawn funds

Description

In `process_withdraw_queue`, the formula for calculating the amount of USDC tokens to return depends on the value of `redeemable_mint.supply`. If the instruction is processing more than two withdrawal requests then `redeemable_mint.supply` will change after the first request gets processed. The change will occur due to the invocation of `token::burn`. Since `token::burn` is a cross-program invocation, the changed value of `redeemable_mint.supply` will not automatically be reflected inside of `process_withdraw_queue`.

This leads to the returned amount of USDC tokens being significantly smaller than what the user would expect based on the specification.

cega-vault/src/lib.rs

```
// Amount in USDC to return to user.
let underlying_amount = get_withdraw_underlying_amount(
     queue_node_to_process.amount,
     ctx.accounts.redeemable_mint.supply,
     ctx.accounts.vault.underlying_amount,
);

let seeds = product_authority_seeds!(bump =
ctx.accounts.state.product_authority_nonce);
// Burn the user's redeemable tokens.
token::burn(
     ctx.accounts.into_burn_context(&[&seeds[..]]),
     queue_node_to_process.amount,
)?;
```

cega-vault/src/utils.rs

```
// Calculate underlying tokens to return based on fraction of
redeemable tokens user owns.
pub fn get_withdraw_underlying_amount(
      redeemable_amount: u64,
      redeemable_amount_supply: u64,
      total_contract_value_at_expiry: u64,
) -> u64 {
      let withdraw_amount = (redeemable_amount as u128)
      .checked_mul(total_contract_value_at_expiry as u128)
      .unwrap()
      .checked_div(redeemable_amount_supply as u128)
      .unwrap();
      return withdraw_amount as u64;
}
```

Additionally, the scenario presented causes redeemable tokens for this vault to be worth more than they are supposed to.. Because of the increase in value, an attacker can withdraw their redeemable tokens at the next rollover and profit.

Proof of Concept

1. Initialize an empty vault with all fees and APR percentages set to zero.
2. User A deposits 10,000 USDC to the vault and receives the same amount of redeemable tokens back.
3. User B deposits 30,000 USDC to the vault and receives the same amount of redeemable tokens back.
4. During the first yield period, user B submits a withdrawal request with 10,000 of their redeemable tokens.
5. During the first yield period, user A submits a withdrawal request with all of their redeemable tokens.
6. The withdrawal queue now has user B's request as the first node and user A's request as the second.
7. The `process_withdraw_queue` instruction is called with both queue nodes.
    a. User B gets their request processed first and gets back 10,000 USDC.
    b. 10,000 redeemable tokens are burned, but `redeemable_mint.supply` stays the same.

   c. User A gets their request processed and gets back 7,500 USDC. (`user_withdraw_amount = redeemables*vault_usdc/redeemable_supply =` 10,000*30,000/40,000 = 7,500)
8. The vault rolls over and subsequently goes through all necessary steps.
9. User B submits a withdrawal request for the remaining redeemable tokens (20,000).
10. `Process_withdraw_queue` is called.
    a. User B gets back 22,500 USDC. (`user_withdraw_amount = redeemables*vault_usdc/redeemable_supply =` 20,000*22,500/20,000 = 22,500)
11. At the end, user B has 32,500 USDC tokens, meaning user B profited.

User B has thus profited at the expense of user A.

Remediation

Call `ctx.accounts.redeemable_mint.reload()` right after the invocation of `token::burn`.

Patch

Reload is now called after the burn invocation. Fixed in #156.

OS-CEG-ADV-02 [Critical] [Resolved]: Missed Knock-in

Description

The `calculation_agent` and `calculate_vault_payoff` instructions do not check the length of `remaining_accounts` passed in to the instruction. The `oracle` and `OptionBarrier` accounts are fetched and deserialized from `remaining_accounts`. These accounts are used to calculate the knock-in ratio if a knock-in event takes place.

If no `remaining_accounts` are passed in, the users are paid fully as per the APR even if a knock-in event took place [1]. This results in a loss of funds for the protocol. The affected code can be seen in the snippet below. Only a modular check [0] is performed on `remaining_accounts.len()`.

cega-vault/src/lib.rs

```
vault.vault_total_coupon_payoff = coupon_payment;
// Calculate principle
// If KI happened -> calculate ratio of spot / strike for each asset
if vault.knock_in_event {
            if ctx.remaining_accounts.len() % 2 != 0 { // [0]
                return Err(error!(

VaultError::IncorrectRemainingAccountsForCalculateVaultPayoff
                ));
            }
            for i in 0..ctx.remaining_accounts.len() / 2 {
[...]
            }
            // Get lowest ratio in worst-off put basket.
            let min_value = option_vector.iter().min();
            msg!("knock in event. Minimum ratio: {:?}",
Some(min_value));
            match min_value {
                Some(min) => {
[...]
                }
                None => {
                    vault.vault_final_payoff =
                        vault.underlying_amount +
vault.vault_total_coupon_payoff; // [1]
                }
```

This bug was introduced in commit a700d61, which was pushed to the repository after we started the audit.

Proof of Concept

1. A vault is set up along with a product.
2. The attacker deposits USDC to the vault and receives redeemable tokens.
3. During the yield period, the attacker places a withdrawal request on queue.
4. A knock-in event takes place during the yield period when admin invokes `calculation_agent`.
5. The yield period concludes and the attacker invokes `calculate_vault_payoff` without any `remaining_accounts`.
6. The withdrawal request gets processed once the `process_withdraw_queue` instruction is invoked.
7. The attacker gets paid fully as per the APR.

Remediation

The program must validate `ctx.remaining_accounts.len()` to match the correct number of `OptionBarrier` accounts for a product. Please refer to the code snippet below for more information.

```
if (ctx.accounts.structured_product_info_account.number_of_puts * 2)
     != ctx.remaining_accounts.len() as u64 {
          return Err(error!(...));
}
```

Patch

Ensure that `ctx.remaining_accounts.len()` is validated correctly. Fixed in #182.

OS-CEG-ADV-03 [Critical] [Resolved]: Insufficient OptionBarrier Checks

Description

The `calculation_agent` and `calculate_vault_payoff` instructions do not validate the `oracle` and `OptionBarrier` accounts. These accounts are fetched and deserialized from `remaining_accounts` and are used to calculate knock-in ratio.

An attacker can use arbitrary `OptionBarrier` and `oracle` accounts to control the knock-in ratio. This would allow the attacker to get paid fully as per the APR even if a knock-in event took place.

The affected code can be found in the snippet below. It can be seen that the program does not perform any validation on the `oracle` and the `OptionBarrier` accounts. `remaining_accounts` are not deserialized or validated by Anchor and must be carefully checked before they are used by the program.

cega-vault/src/lib.rs

```
for i in 0..ctx.remaining_accounts.len() / 2 { // [0]
        let option_barrier: &mut Account<'_, OptionBarrier> =
                &mut Account::try_from(&ctx.remaining_accounts[i *
2].to_account_info())?;
        let oracle = &ctx.remaining_accounts[i * 2 +
1].to_account_info();
        let spot;
        if vault.knock_out_event {
                let price = match option_barrier.last_price {
                        Some(val) => Some(val),
                        None => get_oracle_price(oracle),
                };
                spot = price;
        } else {
                spot = get_oracle_price(oracle);
        }
        option_vector.push(calculate_knock_in_ratio(spot,
option_barrier.strike_abs));
}
// Get lowest ratio in worst-off put basket.
let min_value = option_vector.iter().min();
```

```
match min_value {
        Some(min) => {
                let principle_returned =
                  calculate_principle_return(vault.underlying_amount,
*min);
```

This bug was introduced in commit a700d61, which was pushed to the repository after we started our audit.

Proof of Concept

1. A vault is set up along with a product.
2. Attacker deposits USDC to the vault and receives redeemable tokens.
3. During the yield period, the attacker places a withdrawal request on queue.
4. A knock-in event takes place during the yield period when admin invokes `calculation_agent`.
5. Once the yield period is over, the attacker invokes `calculate_vault_payoff` with an arbitrary `oracle` and `OptionBarrier` accounts such that the knock-in ratio is one.
6. Duplicate `oracle` and `OptionBarrier` accounts can be used for each iteration [0].
7. The withdrawal request gets processed once the `process_withdraw_queue` instruction is invoked.
8. The attacker gets paid fully as per the APR.

Remediation

The program must ensure that all `OptionBarrier` accounts associated with the product are passed in as `remaining_accounts`. It should also check that the `Pubkey` for the passed in `oracle` account is equal to `option_barrier.oracle`.

Patch

Ensure that `OptionBarrier` and `oracle` accounts are validated correctly. Fixed in commit e5360c6.

OS-CEG-ADV-04 [High] [Resolved]: Denial of withdrawal

Description

The instruction `process_withdraw_queue` does not check if all nodes have been removed from the queue before it marks `vault.withdraw_queue_processed=true`. An attacker can arrange their withdrawal request to be the first node on the queue and then submit the `process_withdraw_queue` instruction with only their node in `remaining_accounts`. This will set `vault.withdraw_queue_processed` to `true` which will block all the other nodes already on the withdrawal queue from being processed.

cega-vault/src/lib.rs

```
if ctx.accounts.vault.withdraw_queue_processed {
     return Err(error!(
     VaultError::WithdrawQueueAlreadyProcessedForThisEpoch
     ));
}
[...]
for i in 0..ctx.remaining_accounts.len() / 3 {
    [...]
}
let vault = &mut ctx.accounts.vault;
vault.withdraw_queue_processed = true;
return Ok(());
```

Proof of Concept

1. Set up the vault.
2. User A deposits USDC into the vault.
3. User B deposits a small amount of USDC into the vault.
4. During the yield period, user B places a withdraw request in the queue.
5. During the yield period, user A places a withdraw request in the queue.
6. User B waits for admin to invoke the `collect_fees` instruction.
7. Right after invocation, user B invokes the `process_withdraw_queue` instruction with only their account in `remaining_accounts`.
    a. User B gets their small amount of USDC back.
    b. `Vault.withdraw_queue_processed` is set to `true`.
8. User A is unable to get their USDC back even though the withdrawal request was submitted during the yield period.

If a knock-in takes place before the next rollover, user A loses funds.

Remediation
Check that all nodes have been processed before updating `vault.withdraw_queue_processed`.

Patch
The queue size is checked before setting `vault.withdraw_queue_processed`. Additionally, `process_withdraw_queue` now requires admin signature . Fixed in #156.

OS-CEG-ADV-05 [High] [Resolved]: Vault denial of service

Description

In the `rollover_vault` instruction there is an update to `vault.max_deposit_limit`. Looking closely at this calculation, it was discovered that it could panic as a result of subtraction overflow.

cega-vault/src/lib.rs

```
// Update deposit limit
let deposit_limit = ctx
      .accounts
      .product
      .max_deposit_limit
      .checked_sub(ctx.accounts.product.underlying_amount)
      .unwrap();
if deposit_limit <= 0 {
      vault.max_deposit_limit = 0;
} else {
      vault.max_deposit_limit = deposit_limit;
}
vault.rollover();
Ok(())
```

It is possible for `product.underlying_amount` to become more than `product.max_deposit_limit`. Inside `deposit_vault`, the only checks performed are per-vault, but since a product can be associated with multiple vaults, it is possible for the sum of their `underlying_amount` values to be more than `product.max_deposit_limit`.

cega-vault/src/lib.rs

```
 pub fn deposit_vault(ctx: Context<DepositVault>, args:
DepositVaultArgs) -> Result<()> {
    [...]
      // Check if max vault limit has already been reached
      if ctx.accounts.vault.underlying_amount >=
ctx.accounts.vault.max_deposit_limit {
            return Err(error!(VaultError::MaxDepositLimitReached));
      }
```

```
        // Calculate how much of user's deposit we can accept
        let user_deposit_amount = get_underlying_deposit_amounts(
              args.underlying_amount,
              ctx.accounts.vault.max_deposit_limit,
              ctx.accounts.vault.underlying_amount,
        );

        if user_deposit_amount != 0 {
      [...]
             product.underlying_amount = product
                   .underlying_amount
                   .checked_add(user_deposit_amount)
                   .unwrap();
      [...]
```

Additionally, `underlying_amount` can increase during the yield period due to payoffs. Overflow would result in denial of service for all the vaults associated with this product. Any time an admin tries to call `rollover_vault`, the program will panic and fail the instruction. Since rollover is not possible, , all the boolean flags which track the vaults' states will never be reset. Hence, the vaults can't progress.

cega-vault/src/account.rs

```
pub fn rollover(&mut self) {
      self.epoch_sequence_number =
self.epoch_sequence_number.checked_add(1).unwrap();
      self.knock_in_event = false;
      self.knock_out_event = false;
      self.trade_processed = false;
      self.fees_collected = false;
      self.payoff_calculated = false;
      self.withdraw_queue_processed = false;
      self.vault_final_payoff = 0;
      self.vault_total_coupon_payoff = 0;
}
```

The chances of this occurring depend on how high `product.max_deposit_limit` will be, how many vaults will be associated with a single product, and whether users are able to reach `max_deposit_limit`.

Proof of Concept

1. Initialize two vaults associated with a single product.
    a. `vault.max_deposit_limit` is set to `product.max_deposit_limit` for both vaults, as `product.underlying_amount` is zero initially.
2. User A deposits USDC tokens to the first vault with the amount of tokens being close to the deposit limit.
3. User B deposits USDC tokens to the second vault with the amount of tokens being close to the deposit limit.
4. `product.underlying_amount` is now above `product.max_deposit_limit`.
5. During the yield period, `underlying_amount` increases even more.
6. When admin calls `rollover_vault`, `checked_sub` and `unwrap` lead to panic.
7. As both vaults are unable to progress, users cannot withdraw their funds.

Remediation

Remove the calculation for `vault.max_deposit_limit` update as it does not work as intended. Also, checks inside `deposit_vault` instruction should be performed using `product.underlying_amount` instead of `vault.underlying_amount.`

Patch

`max_deposit_limit` has been removed entirely from the Vault account struct. Code for overflowing subtraction is removed as well. Fixed in commit 0669889.

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-CEG-SUG-00	Pyth oracle options price update is missing confidence interval checks.
OS-CEG-SUG-01	Denial of service in calculation agent due to an integer overflow when calculating price from Pyth.
OS-CEG-SUG-02	TokenAccount owner not checked after deserializing from remaining accounts .
OS-CEG-SUG-03	Potential seed collision on product account.
OS-CEG-SUG-04	Denial of service in calculation agent due to insufficient checks while updating observation time.
OS-CEG-SUG-05	Missed vulnerabilities due to low test coverage.
OS-CEG-SUG-06	Credit risk during yield generation.

OS-CEG-SUG-00: Missing Pyth Oracle Confidence Interval Check

Description

The program is missing confidence interval checks, while fetching asset price updates for options from Pyth's oracle.

A high confidence interval denotes that Pyth's price is inaccurate and must not be used by the program. Alternatively, the program can fall back to another oracle. It can be seen in the following snippet that the program uses Pyth's price without checking `current_price.conf`.

cega-vault/src/utils.rs

```
let current_price: Price = price_feed.get_current_price().unwrap();
let price = (current_price.price as u128)
    .checked_mul(10u128.pow(USDC_DECIMALS))
    .unwrap()
    .checked_div(10u128.pow((-current_price.expo) as u32))
    .unwrap() as u128;
return Some(price);
```

Remediation

It is recommended to add a confidence interval check while using the Pyth oracle[1]. For more information, please refer to the code linked in the footnote below.

mango-v3/program/src/processor.rs

```
if conf > PYTH_CONF_FILTER {
    msg!(
        "Pyth conf interval too high; oracle index: {} value: {}
conf: {}", [...]
    );
    return Err(throw_err!(MangoErrorCode::InvalidOraclePrice));
}
```

[1] https://github.com/blockworks-foundation/mango-v3/blob/6b01e3f63d2a0f97eb08b734317ef47e3f667c4f/program/src/processor.rs#L6860

Patch

The confidence interval is now checked. Fixed in #198.

OS-CEG-SUG-01: Pyth Oracle Integer Overflow

Description

In `get_oracle_price`, the program assumes that the price exponent will be negative [0]. Unfortunately, the Pyth oracle program allows for the price exponent of a product to be positive.

pyth-client/program/src/oracle/oracle.c

```
static uint64_t add_price( SolParameters *prm, SolAccountInfo *ka )
{
  // Validate command parameters
  cmd_add_price_t *cptr = (cmd_add_price_t*)prm->data;
  if ( prm->data_len != sizeof( cmd_add_price_t ) ||
       cptr->expo_ > PC_MAX_NUM_DECIMALS ||
       cptr->expo_ < -PC_MAX_NUM_DECIMALS ||
```

cega-vault/src/utils.rs

```
let price = (current_price.price as u128)
    .checked_mul(10u128.pow(USDC_DECIMALS))
    .unwrap()
    .checked_div(10u128.pow((-current_price.expo) as u32)) // [0]
    .unwrap() as u128;
```

If `current_price.expo` is positive, the pow operation will lead to an integer overflow and the program will panic. Due to this, all calls to `calculation_agent` will panic. This isn't necessarily an issue because all products currently listed on Pyth have a negative price exponent. However, the program must perform the price calculation correctly to avoid any future implications.

Remediation

Check if `current_price.expo` is positive, and use `checked_mul` in such a case. For more information, please refer to the code snippet below.

```
let mut price = (current_price.price as u128)
    .checked_mul(10u128.pow(USDC_DECIMALS))
    .unwrap();
```

```
if current_price.expo < 0 {
    price = price.checked_div(10u128.pow((-current_price.expo) as
u32)).unwrap() as u128;
} else {
    price = price.checked_mul(10u128.pow(current_price.expo as
u32)).unwrap() as u128;
}
return Some(price);
```

Patch

The issue has been fixed and now `checked_mul` is used if `current_price.expo` is positive. Fixed in #194.

OS-CEG-SUG-02: Explicitly Check Remaining Accounts

Description

In the `process_withdraw_queue` instruction, TokenAccounts are fetched and deserialized from `remaining_accounts[0]`. Remaining accounts are not validated by anchor and must be used carefully.

The `TokenAccount` is deserialized directly with borrowed account data, and it's not checked if the account is owned by the token program. This isn't a vulnerability as the `token::transfer` invocation will fail if the `TokenAccount` is not owned by the token program. However, it is recommended to check the account owner explicitly when using `remaining_accounts`.

cega-vault/src/lib.rs

```
let user_underlying_token_account_to_process: &mut TokenAccount =
    &mut TokenAccount::try_deserialize(
    &mut &ctx.remaining_accounts[i * 3 + 1].try_borrow_data()?[..],
    )
    .unwrap(); // [0]

let user_authority = ctx.remaining_accounts[i * 3 +
2].to_account_info();
[...]
if user_underlying_token_account_to_process.owner !=
user_authority.key() {
    return
Err(error!(VaultError::IncorrectUserAuthorityForTokenAccount));
}
```

Remediation

Use `get_associated_token_address` to derive the address of an associated token account. Avoid using `remaining_accounts` when possible.

OS-CEG-SUG-03: Product Account Seed Collision

Description

It was noticed that only the `product.product_name` was being used to derive the product account seed. It is recommended to prefix the seed with a constant, to avoid seed collision with other accounts.

cega-vault/src/context.rs

```
#[account(
    mut,
    seeds = [product.product_name.as_ref().strip()],
    bump = product.product_nonce,
)]
pub product: Box<Account<'info, Product>>,
```

Remediation

Add a prefix before `product.product_name`. Please refer to the code snippet below for more information.

```
#[account(
    mut,
    seeds = [PRODUCT_SEED, product.product_name.as_ref().strip()],
    bump = product.product_nonce,
)]
pub product: Box<Account<'info, Product>>,
```

Patch

All `Product` account seeds now have `PRODUCT_SEED` constant as prefix. Fixed in #196.

OS-CEG-SUG-04: Override observation period validation

Description

The `override_observation_period` instruction fails to ensure that `args.observation_time` is valid. This can lead to a denial of service in the calculation agent, if `args.observation_time` is from the past.

cega-vault/src/lib.rs

```rust
pub fn override_observation_period(
        ctx: Context<OverrideObservationPeriod>,
        args: OverrideObservationPeriodArgs,
    ) -> Result<()> {
        let option_barrier = &mut ctx.accounts.option_barrier;
        [...]
        option_barrier.observation_time = args.observation_time;
```

cega-vault/src/lib.rs

```rust
let lower_time_bound = option_barrier
    .observation_time
    .checked_sub(option_barrier.time_buffer)
    .unwrap();
let upper_time_bound = option_barrier
    .observation_time
    .checked_add(option_barrier.time_buffer)
    .unwrap();
if (clock.unix_timestamp as u64) < lower_time_bound
    || (clock.unix_timestamp as u64) > upper_time_bound
{
    return Err(error!(
        VaultError::InvalidObservationPeriodForCalculationAgent
    ));
}
```

Remediation

Ensure `args.observation_time` is valid. For more information please refer to the code snippet below.

```
pub fn override_observation_period(
        ctx: Context<OverrideObservationPeriod>,
        args: OverrideObservationPeriodArgs,
    ) -> Result<()> {
        let option_barrier = &mut ctx.accounts.option_barrier;
        let clock = Clock::get()?;
        if (clock.unix_timestamp as u64) >= args.observation_time {
            return Err(error!(...));
        }
[...]
        option_barrier.observation_time = args.observation_time;
```

Patch

The issue has been fixed and now if `args.observation_time` is less than `clock.unix_timestamp`, the instruction returns an error. Fixed in #195.

OS-CEG-SUG-05: Low Test Coverage

Description
While testing, it was noticed that the test suite for the program does not provide enough coverage. Multiple vulnerabilities found throughout this audit could've been caught with more coverage. For example OS-CEG-ADV-01, results in significantly lower payouts for most users. This would've been found with a test simulating a withdrawal from multiple users during the yield period, and comparing the amount with the expected APR.

Another example is OS-CEG-ADV-05; wherein there could be a denial of service for all vaults associated with a product. A simple test which deposits USDC tokens to multiple vaults for a product could've triggered this vulnerability.

Remediation
Ideally, the test suite should be equipped with multiple test cases that provide coverage for the whole program. The test cases should simulate possible real world scenarios, and ensure that the program functionality is sound.

Important scenarios to test:

- Knock In/Knock Out events
- Multiple users
- Multiple vaults

To model complex scenarios, tests will require sources like clock and price oracles to be mocked. This is especially important for Cega Vault since its operation heavily depends on time.

OS-CEG-SUG-06: Yield generation credit risk

Description
In a scenario where no knock-in events take place for a product, the payout is solely dependent on yield from maple. If maple fails to generate the yield as per the APR, cega vault will not have enough liquidity to payout users. In addition to the market risk, this also exposes cega users to credit risk if the market makers that cega trades with default.

Remediation
Ideally, the knock-in levels and quoted APR rates for products should be chosen carefully to minimize investors exposure to credit risk.

07 | Appendix

Appendix A: Program Files

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

```
cega-vault
  Cargo.toml                                5c32caf68f0ee876f5679430d4d471d8
  Xargo.toml                                815f2dfb6197712a703a8e1f75b03c69
  src
    account.rs                              ba9bb5bd75c60b8dc490cc09f9a1efd3
    address.rs                              7bce0792baf7da1714f76ef31f28d1a9
    constants.rs                            96f68755c150489756f1e08416e12e70
    context.rs                              163b3ec00adc14dfb337677a6bb139a9
    errors.rs                               222b047dae14e178d2cf9528dc041fa7
    events.rs                               df3b973cc686ee4847796120989f6ef7
    lib.rs                                  4dd4fd39db5c36b9892d60ae51ecc8c7
    notes.txt                               5b72e78465a60e64d724703ed31ffe92
    tests.rs                                8ddbed3ac4d299236024259e2dc45df8
    utils.rs                                22041be3391208eaee518bda27eb113f
    validation.rs                           7df3504fc260e7218ca9ed1cbe39a399
```

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.

Appendix C: Proofs of Concept

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

```
Dockerfile                                f365a6e9e22ec3cdf6d98fc11aa47509
README.md                                 5fb2a072845d599414bb1ec62b9a2aad
run.sh                                    1d0e9bf3949fa05be8847d03b03c0de7
config
  run.sh                                  45c6c0a63539a88505238a29cf1be5e3
pocs
  os-ceg-adv-00
      hash                                ac068731e9396e49014743e420cbe187
      patch                               59918f4d15d7598ca97cf1d560ddab30
      run.sh                              e3e13cb1bea31573986f7e1d86dc627c
  os-ceg-adv-01
      hash                                ac068731e9396e49014743e420cbe187
      patch                               e44f65e9e9d0cc996f47460536ddcfe5
      run.sh                              ed6872c981c5003db8303288f912a706
  os-ceg-adv-02
      hash                                a2392c67474182735700701020907 83e
      patch                               4f47abf45c0c4c0b05f1134dd7882842
      run.sh                              6e05eb6b3481358c3f280d20d316e1f3
  os-ceg-adv-03
      hash                                99d82b030b357844a1a6c08e1756029c
      patch                               79238032873e4a32a274c7011efdcf5b
      run.sh                              2dc4da9d11c24e5404fba82c22aca4c1
  os-ceg-adv-04
      hash                                ac068731e9396e49014743e420cbe187
      patch                               965c26cf35715f6fcd9746e478d6d012
      run.sh                              829eadc5a7f6b3b829888b879500928d
  os-ceg-adv-05
      hash                                ac068731e9396e49014743e420cbe187
      patch                               d04e07ce4f19acc8380ee95183170ce4
      run.sh                              a7d038bc7502ac30462f6e11a5386879
```

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)