FYEO

Security Assessment of the Bitgreen Node

Bitgreen, Inc.

January 2023
Version 1.0

Presented by:

FYEO Inc.

PO Box 147044
Lakewood CO 80214
United States

Security Level
Strictly Confidential

Table of Contents

List of Figures

LIST OF TABLES

EXECUTIVE SUMMARY

Overview

Bitgreen, Inc. engaged FYEO Inc. to perform a Security Assessment of the Bitgreen Node.

The assessment was conducted remotely by the FYEO Security Team. Testing took place on December 12 - December 16, 2022, and focused on the following objectives:

- To provide the customer with an assessment of their overall security posture and any risks that were discovered within the environment during the engagement.
- To provide a professional opinion on the maturity, adequacy, and efficiency of the security measures that are in place.
- To identify potential issues and include improvement recommendations based on the results of our tests.

This report summarizes the engagement, tests performed, and findings. It also contains detailed descriptions of the discovered vulnerabilities, steps the FYEO Security Team took to identify and validate each issue, as well as any applicable recommendations for remediation.

Key Findings

The following issues were identified during the testing period. They have since been remediated:

- FYEO-BG-01 – Stable coins like aUSD have many digits; cast to u32 may stop program
- FYEO-BG-02 – Maximum fee a user is willing to pay is not a parameter
- FYEO-BG-03 – OrderId as u64 may be too little
- FYEO-BG-04 – Flooring fee calculation
- FYEO-BG-05 – PaymentFees and PurchaseFees have no limit (0-100%)
- FYEO-BG-06 – The creator of a sell order can buy from themselves
- FYEO-BG-07 – Copy & Paste issue with the Documentation

Based on our review process, we conclude that the reviewed code implements the documented functionality.

Scope and Rules of Engagement

The FYEO Review Team performed a Security Assessment of the Bitgreen Node. The following table documents the targets in scope for the engagement. No additional systems or resources were in scope for this assessment.

The source code was supplied through a public repository at https://github.com/bitgreen/bitgreen-node/tree/release with the commit hash 4e1620b81acdc31c2e770189c61044a548d6744d.

A re-review of the code was carried out and issues remediated with the commit hash 879eb96901963012338774ef2b08252e608778f0

Files included in the code review
```
node/
├── pallets/

│   ├── dex/
│   │   ├── src/
│   │   │   ├── benchmarking.rs
│   │   │   ├── lib.rs
│   │   │   ├── mock.rs
│   │   │   └── tests.rs
│   │   └── Cargo.toml
```

Table 1: Scope

TECHNICAL ANALYSES AND FINDINGS

During the Security Assessment of the Bitgreen Node, we discovered:

- 1 finding with HIGH severity rating.
- 2 findings with MEDIUM severity rating.
- 3 findings with LOW severity rating.
- 1 finding with INFORMATIONAL severity rating.

The following chart displays the findings by severity.

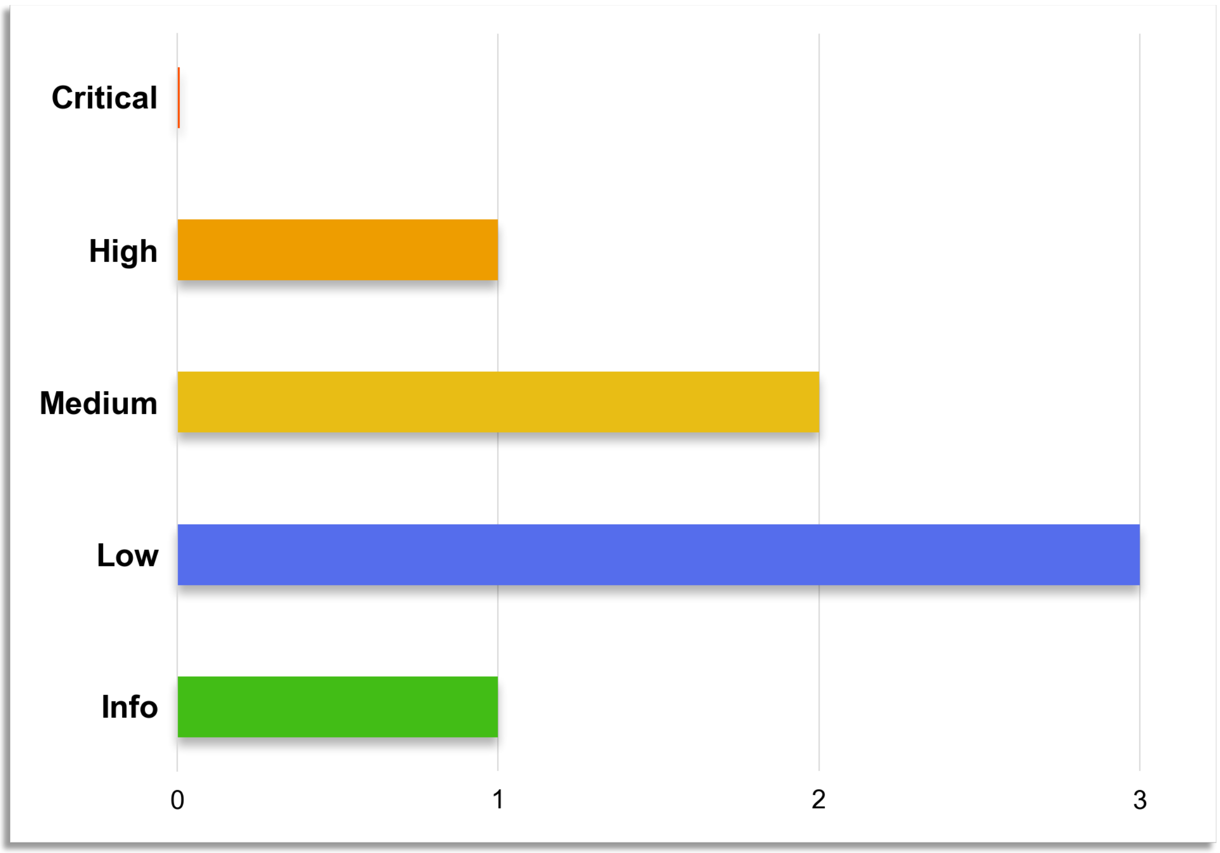

Figure 1: Findings by Severity

Findings

The *Findings* section provides detailed information on each of the findings, including methods of discovery, explanation of severity determination, recommendations, and applicable references.

The following table provides an overview of the findings.

Finding #	Severity	Description
FYEO-BG-01	High	Stable coins like aUSD have many digits; cast to u32 may stop program
FYEO-BG-02	Medium	Maximum fee a user is willing to pay is not a parameter
FYEO-BG-03	Medium	OrderId as u64 may be too little
FYEO-BG-04	Low	Flooring fee calculation
FYEO-BG-05	Low	PaymentFees and PurchaseFees have no limit (0-100%)
FYEO-BG-06	Low	The creator of a sell order can buy from themselves
FYEO-BG-07	Informational	Copy & Paste issue with the Documentation

Table 2: Findings Overview

Technical Analysis

The source code has been manually validated to the extent that the state of the repository allowed. The validation includes confirming that the code correctly implements the intended functionality.

Technical Findings

General Observations

During code assessment, it was noted that the rust code is very well written and structured. The use of checked arithmetic operations to protect from overflow/underflow operations shows commitment to writing secure programs.

User-supplied parameters are thoroughly checked for edge cases such as 0 amounts.

The importance of having minimums for things like prices and units sold was also recognized, and the developers have implemented the appropriate checks.

The in-code documentation and comments were also well done and give a good insight into what is going on.

STABLE COINS LIKE AUSD HAVE MANY DIGITS; CAST TO U32 MAY STOP PROGRAM

Finding ID: FYEO-BG-01
Severity: High
Status: Remediated

Description

The code base appears to assume that an integer value of 1 is equal to 1 USD of some stablecoin. Most tokens, including stablecoins, have 10 or even 12 decimals. The code currently seems to implement Acala's aUSD, which has 12 digits meaning 1 USD = 1 000 000 000 000 tokens.

The code casts the price per unit into a u32, which is limited to a value of 4 294 967 295 - a number only big enough to represent fractions of a USD.

This would imply the price would be restricted to fractions of dollars, but carbon tons are presumably more expensive. Any price per carbon ton greater than ~0.004 USD would throw an ArithmeticError and abort the transaction.

Assuming a token with 6 decimals this would still restrict the price per unit to 4294 USD. With a price per unit (per carbon ton) of 75 USD that would mean any trade of more than 57 units would break the program.

```
75 000 000 * 58 = 4 350 000 000 -> ArithmeticError
```

Proof of Issue

File name: pallets/dex/src/lib.rs
Line number: 306

```
let price_per_unit_as_u32: u32 =
    order.price_per_unit.try_into().map_err(|_| Error::<T>::ArithmeticError)?;

let required_currency = price_per_unit_as_u32
    .checked_mul(units_as_u32)
    .ok_or(Error::<T>::ArithmeticError)?;
```

Severity and Impact Summary

If the stablecoin is not somehow scaled to fewer decimals, the program can't function. If it is scaled, the flooring transaction fee issue becomes a more severe problem.

Recommendation

Consider how many decimal places the stablecoin used has and adjust the code accordingly.

References

https://wiki.acala.network/integrate/acala/protocol-info
https://statemint.subscan.io/assets/1984

Maximum fee a user is willing to pay is not a parameter

Finding ID: FYEO-BG-02
Severity: Medium
Status: Remediated

Description

It is possible that the user sends a buy order and the admin changes the fee to 100% PaymentFees + 100% PurchaseFees = 200% of the total order volume. If someone buys credits for a million USD they could be agreeing to pay a 2 million USD fee, if the admin's update fee transactions settle first.

Proof of Issue

File name: pallets/dex/src/lib.rs
Line number: 278

```
pub fn buy_order(
    origin: OriginFor<T>,
    order_id: OrderId,
    asset_id: AssetIdOf<T>,
    units: AssetBalanceOf<T>,
) -> DispatchResult {
    ...
}
```

Severity and Impact Summary

The function forces the user to agree to whatever fee is set when the transaction settles.

Recommendation

The expected fee should be parameterized and sent by the front end. If that's not enough to satisfy currently configured fees, the TX should fail.

ORDERID AS U64 MAY BE TOO LITTLE

Finding ID: FYEO-BG-03
Severity: Medium
Status: Remediated

Description

The order id is incremented by a u64 counter. This counter limits the total number of orders that can ever be created. While an u64 can hold a huge number, There is no loss (other than TX fees) for a user creating and cancelling orders. If it were possible to create millions of orders (DoS attack) for only cents in fees, users could considerably ‘shrink’ the u64 space or make it run out of IDs entirely.

Proof of Issue

File name: pallets/dex/src/lib.rs
Line number: 54

```
pub type OrderId = u64;
```

Severity and Impact Summary

The severity depends on the actual TX fee cost for creating and cancelling an order. If it becomes too cheap, the u64 space may be filled up just by creating and cancelling dummy orders, making it impossible to create additional orders.

Recommendation

Make sure the TX fee is high enough to prevent such user behaviour. It would also be an option to consider adding a fee for listing or cancelling orders.

FLOORING FEE CALCULATION

Finding ID: FYEO-BG-04
Severity: Low
Status: Remediated

Description

Assuming a 75 price, and both fees at 10% each; if a user buys 50 units they'll pay 750 in fees If they instead have 50 separate orders of 1 unit each, they'll pay 700 in fees. So they can save 50 in fees doing that - cause of the flooring calculation.

Proof of Issue

File name: pallets/dex/src/lib.rs
Line number: 312

```
let payment_fee = PaymentFees::<T>::get().mul_floor(required_currency);
let purchase_fee = PurchaseFees::<T>::get().mul_floor(required_currency);
```

Severity and Impact Summary

There could be some incentive for users to create more orders rather than fewer. This would also depend on the network fees.

The severity of this issue also depends on how many digits the stablecoin uses.

Recommendation

Consider adjusting the calculation.

PaymentFees and PurchaseFees have no limit (0-100%)

Finding ID: FYEO-BG-05
Severity: Low
Status: Remediated

Description

Both the PaymentFees and PurchaseFees have no limit (0-100%). They may add up to 200% of the purchase price.

Proof of Issue

File name: pallets/dex/src/lib.rs
Line number: 358

```
pub fn force_set_payment_fee(origin: OriginFor<T>, payment_fee: Percent) ->
DispatchResult {
    T::ForceOrigin::ensure_origin(origin)?;
    PaymentFees::<T>::set(payment_fee);
    Ok(())
}
```

File name: pallets/dex/src/lib.rs
Line number: 368

```
pub fn force_set_purchase_fee(origin: OriginFor<T>, purchase_fee: Percent) ->
DispatchResult {
    T::ForceOrigin::ensure_origin(origin)?;
    PurchaseFees::<T>::set(purchase_fee);
    Ok(())
}
```

Severity and Impact Summary

There are no limits to the fees and they may add up to 200% of the purchase price.

Recommendation

Consider choosing a hard limit on fees.

THE CREATOR OF A SELL ORDER CAN BUY FROM THEMSELVES

Finding ID: FYEO-BG-06
Severity: Low
Status: Remediated

Description

In the `buy_order` function there is no check if the `buyer == order.owner`.

Proof of Issue

File name: pallets/dex/src/lib.rs
Line number: 278

```
ensure!(asset_id == order.asset_id, Error::<T>::InvalidAssetId);

// ensure volume remaining can cover the buy order
ensure!(units <= order.units, Error::<T>::OrderUnitsOverflow);

...
```

Severity and Impact Summary

People would presumably not want to buy their own orders, since the buyer is paying a fee, but if they do, they will incur a fee and thus have some loss.

Recommendation

Unless this is intentional it would be best to prevent this case.

Copy & Paste issue with the Documentation

Finding ID: FYEO-BG-07
Severity: Informational
Status: Remediated

Description

There is a mistake in the in-code documentation for the `create_sell_order` function:

Proof of Issue

File name: pallets/dex/src/lib.rs
Line number: 17

```
//! * `create_sell_order`: Creates a new project onchain with details of
batches of credits
```

Severity and Impact Summary

No impact.

Recommendation

It would be good to keep the documentation in order.

OUR PROCESS

METHODOLOGY

FYEO Inc. uses the following high-level methodology when approaching engagements. They are broken up into the following phases.

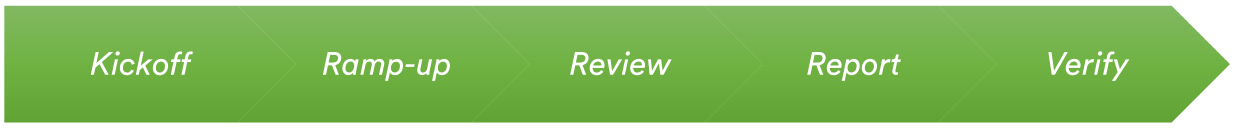

Figure 2: Methodology Flow

KICKOFF

The project is kicked off as the sales process has concluded. We typically set up a kickoff meeting where project stakeholders are gathered to discuss the project as well as the responsibilities of participants. During this meeting we verify the scope of the engagement and discuss the project activities. It's an opportunity for both sides to ask questions and get to know each other. By the end of the kickoff there is an understanding of the following:

- Designated points of contact
- Communication methods and frequency
- Shared documentation
- Code and/or any other artifacts necessary for project success
- Follow-up meeting schedule, such as a technical walkthrough
- Understanding of timeline and duration

RAMP-UP

Ramp-up consists of the activities necessary to gain proficiency on the project. This can include the steps needed for familiarity with the codebase or technological innovation utilized. This may include, but is not limited to:

- Reviewing previous work in the area including academic papers
- Reviewing programming language constructs for specific languages
- Researching common flaws and recent technological advancements

Review

The review phase is where most of the work on the engagement is completed. This is the phase where we analyze the project for flaws and issues that impact the security posture. Depending on the project this may include an analysis of the architecture, a review of the code, and a specification matching to match the architecture to the implemented code.

In this code audit, we performed the following tasks:

1. Security analysis and architecture review of the original protocol
2. Review of the code written for the project
3. Compliance of the code with the provided technical documentation

The review for this project was performed using manual methods and utilizing the experience of the reviewer. No dynamic testing was performed, only the use of custom-built scripts and tools were used to assist the reviewer during the testing. We discuss our methodology in more detail in the following sections.

Code Safety

We analyzed the provided code, checking for issues related to the following categories:

- General code safety and susceptibility to known issues
- Poor coding practices and unsafe behavior
- Leakage of secrets or other sensitive data through memory mismanagement
- Susceptibility to misuse and system errors
- Error management and logging

This list is general and not comprehensive, meant only to give an understanding of the issues we are looking for.

Technical Specification Matching

We analyzed the provided documentation and checked that the code matches the specification. We checked for things such as:

- Proper implementation of the documented protocol phases
- Proper error handling
- Adherence to the protocol logical description

Reporting

FYEO Inc. delivers a draft report that contains an executive summary, technical details, and observations about the project.

The executive summary contains an overview of the engagement including the number of findings as well as a statement about our general risk assessment of the project. We may conclude that the overall risk is low but depending on what was assessed we may conclude that more scrutiny of the project is needed.

We report security issues identified, as well as informational findings for improvement, categorized by the following labels:

- Critical
- High
- Medium
- Low
- Informational

The technical details are aimed more at developers, describing the issues, the severity ranking and recommendations for mitigation.

As we perform the audit, we may identify issues that aren't security related, but are general best practices and steps that can be taken to lower the attack surface of the project. We will call those out as we encounter them and as time permits.

As an optional step, we can agree on the creation of a public report that can be shared and distributed with a larger audience.

Verify

After the preliminary findings have been delivered, this could be in the form of the approved communication channel or delivery of the draft report, we will verify any fixes within a window of time specified in the project. After the fixes have been verified, we will change the status of the finding in the report from open to remediated.

The output of this phase will be a final report with any mitigated findings noted.

Additional Note

It is important to note that, although we did our best in our analysis, no code audit or assessment is a guarantee of the absence of flaws. Our effort was constrained by resource and time limits along with the scope of the agreement.

While assessing the severity of the findings, we considered the impact, ease of exploitability, and the probability of attack. This is a solid baseline for severity determination.

THE CLASSIFICATION OF VULNERABILITIES

Security vulnerabilities and areas for improvement are weighted into one of several categories using, but is not limited to, the criteria listed below:

Critical – vulnerability will lead to a loss of protected assets

- This is a vulnerability that would lead to immediate loss of protected assets
- The complexity to exploit is low
- The probability of exploit is high

High - vulnerability has potential to lead to a loss of protected assets

- All discrepancies found where there is a security claim made in the documentation that cannot be found in the code
- All mismatches from the stated and actual functionality
- Unprotected key material
- Weak encryption of keys
- Badly generated key materials
- Txn signatures not verified
- Spending of funds through logic errors
- Calculation errors overflows and underflows

Medium - vulnerability hampers the uptime of the system or can lead to other problems

- Insecure calls to third party libraries
- Use of untested or nonstandard or non-peer-reviewed crypto functions
- Program crashes, leaves core dumps or writes sensitive data to log files

Low – vulnerability has a security impact but does not directly affect the protected assets

- Overly complex functions
- Unchecked return values from 3rd party libraries that could alter the execution flow

Informational

- General recommendations