Audit Report

Out GCC

v1.0

September 23, 2024

Table of Contents

License

Disclaimer

THE CONTENT OF THIS AUDIT REPORT IS PROVIDED “AS IS”, WITHOUT REPRESENTATIONS AND WARRANTIES OF ANY KIND.

THE AUTHOR AND HIS EMPLOYER DISCLAIM ANY LIABILITY FOR DAMAGE ARISING OUT OF, OR IN CONNECTION WITH, THIS AUDIT REPORT.

THIS AUDIT REPORT WAS PREPARED EXCLUSIVELY FOR AND IN THE INTEREST OF THE CLIENT AND SHALL NOT CONSTRUE ANY LEGAL RELATIONSHIP TOWARDS THIRD PARTIES. IN PARTICULAR, THE AUTHOR AND HIS EMPLOYER UNDERTAKE NO LIABILITY OR RESPONSIBILITY TOWARDS THIRD PARTIES AND PROVIDE NO WARRANTIES REGARDING THE FACTUAL ACCURACY OR COMPLETENESS OF THE AUDIT REPORT.

FOR THE AVOIDANCE OF DOUBT, NOTHING CONTAINED IN THIS AUDIT REPORT SHALL BE CONSTRUED TO IMPOSE ADDITIONAL OBLIGATIONS ON COMPANY, INCLUDING WITHOUT LIMITATION WARRANTIES OR LIABILITIES.

This audit has been performed by

Oak Security GmbH

https://oaksecurity.io/
info@oaksecurity.io

Introduction

Purpose of This Report

Oak Security GmbH has been engaged by MJEO REKLAMCILIK VE TASARIM TICARET LiMiTED SiRKETi to perform a security audit of Out GCC

The objectives of the audit are as follows:

1. Determine the correct functioning of the protocol, in accordance with the project specification.
2. Determine possible vulnerabilities, which could be exploited by an attacker.
3. Determine smart contract bugs, which might lead to unexpected behavior.
4. Analyze whether best practices have been applied during development.
5. Make recommendations to improve code safety and readability.

This report represents a summary of the findings.

As with any code audit, there is a limit to which vulnerabilities can be found, and unexpected execution paths may still be possible. The author of this report does not guarantee complete coverage (see disclaimer).

Codebase Submitted for the Audit

The audit has been performed on the following target:

Repository	https://github.com/stephanOUT/gcc/
Commit	`8e204266fed535cd52e1027bc14151072d33549b`
Scope	All contracts were in scope.
Fixes verified at commit	`bf48bbaf6fbaa92ea7ce0b731e30431a005e5acf` Note that only fixes to the issues described in this report have been reviewed at this commit. Any further changes such as additional features have not been reviewed.

Methodology

The audit has been performed in the following steps:

1. Gaining an understanding of the code base's intended purpose by reading the available documentation.
2. Automated source code and dependency analysis.
3. Manual line-by-line analysis of the source code for security vulnerabilities and use of best practice guidelines, including but not limited to:
    a. Race condition analysis
    b. Under-/overflow issues
    c. Key management vulnerabilities
4. Report preparation

Functionality Overview

The Out GCC program provides each user with a unique token, generated and supplied by a centralized entity based on their social media followers.

Users buy tokens at a fixed price, with a part of the proceeds going to the token's owner and the rest to the platform.

After the initial supply sells out, tokens can be traded on a marketplace, where transaction fees are distributed among the seller, token owner, and platform.

How to Read This Report

This report classifies the issues found into the following severity categories:

Severity	Description
Critical	A serious and exploitable vulnerability that can lead to loss of funds, unrecoverable locked funds, or catastrophic denial of service.
Major	A vulnerability or bug that can affect the correct functioning of the system, lead to incorrect states or denial of service.
Minor	A violation of common best practices or incorrect usage of primitives, which may not currently have a major impact on security, but may do so in the future or introduce inefficiencies.
Informational	Comments and recommendations of design decisions or potential optimizations, that are not relevant to security. Their application may improve aspects, such as user experience or readability, but is not strictly necessary. This category may also include opinionated recommendations that the project team might not share.

The status of an issue can be one of the following: **Pending,** **Acknowledged**, or **Resolved**.

Note that audits are an important step to improving the security of smart contracts and can find many issues. However, auditing complex codebases has its limits and a remaining risk is present (see disclaimer).

Users of the system should exercise caution. In order to help with the evaluation of the remaining risk, we provide a measure of the following key indicators: **code complexity**, **code readability**, **level of documentation**, and **test coverage**. We include a table with these criteria below.

Note that high complexity or low test coverage does not necessarily equate to a higher risk, although certain bugs are more easily detected in unit testing than in a security audit and vice versa.

Code Quality Criteria

The auditor team assesses the codebase's code quality criteria as follows:

Criteria	Status	Comment
Code complexity	Low-Medium	-
Code readability and clarity	Medium	Lack of comments in the code explaining business logic
Level of documentation	Low-Medium	The documentation describes generically the flow of the program without technical details
Test coverage	Low	Anchor tests are currently not functional and cannot be executed. Some of the core functionalities of the protocol, like token transfer, are not covered in the test suite.

Summary of Findings

No	Description	Severity	Status
1	Attackers providing incorrect or malicious price feed can buy tokens at an arbitrary price	Critical	Resolved
2	Market can be initialized without seller authorization and before all tokens have been sold	Critical	Resolved
3	Incorrect math leads to underpriced tokens when buying an offer	Critical	Resolved
4	Token transfers will always fail due to incorrect account seed	Critical	Resolved
5	Inconsistent accounting of token decimals throughout the protocol	Critical	Resolved
6	Users can buy tokens from sellers with the “Stage 1” mechanism after the market is initialized	Major	Acknowledged
7	Token holders can create an infinite number of unbacked offers	Major	Resolved
8	Marketplace offers can be reinitialized	Major	Resolved
9	Invalid boolean arithmetic leads to a verification bypass	Major	Resolved
10	Assuming negative sign of Pyth price feed exponent can lead to incorrect token pricing	Major	Resolved
11	Multiple minting is allowed in the `CreateToken` instruction	Minor	Resolved
12	Lack of input validation for the `initialize_program_state` instruction	Minor	Resolved
13	Possible precision loss due to incorrect calculations	Minor	Partially Resolved
14	Missing check allows to set any account as token authority	Minor	Resolved
15	Program state initialization is fully permissionless	Minor	Resolved
16	Holder status is not used	Informational	Resolved
17	Miscellaneous comments	Informational	Partially Resolved

Detailed Findings

1. Attackers providing incorrect or malicious price feed can buy tokens at an arbitrary price

Severity: Critical

The `feed_account`, defined in `programs/gcx/src/instructions/buy_offer.rs:179` and `programs/gcx/src/instructions/initial_buy.rs:137`, is intended to provide the `SOL/USD` price feed from the Pyth oracle.

However, the current implementation does not validate whether the provided `feed_account` is an account owned by the Pyth oracle, nor does it confirm that the account reports the correct price feed.

Consequently, attackers can exploit this vulnerability by providing a different Pyth price feed account or an entirely fake account with incorrect price data. By doing so, they can manipulate the price feed to purchase tokens at an artificially deflated price, leading to potential financial losses for the platform and its users.

For example, a buyer could present a manipulated price feed that proposes a much lower `SOL/USD` price, allowing them to acquire tokens for far less than their true market value.

Recommendation

We recommend explicitly constraining the `feed_account` to the desired Pyth price feed account address in both the aforementioned locations.

Status: Resolved

2. Market can be initialized without seller authorization and before all tokens have been sold

Severity: Critical

The `InitializeMarket` instruction handler in `programs/gcx/src/instructions/create_token.rs` is intended to enable “Stage 2” of token distribution by allowing the token owner to initiate a market for token exchange after all tokens have been sold during “Stage 1”.

However, there is no verification to ensure all tokens have been sold in “Stage 1” before enabling “Stage 2” and the instruction does not implement any authorization mechanism, permitting anyone to execute it.

Consequently, the market can be initialized before the completion of “Stage 1,” disrupting the token distribution process. Additionally, the lack of authorization allows anyone to start a marketplace for tokens without the owner’s consent, potentially leading to unauthorized market creation and disruption of token economics.

Recommendation

We recommend implementing an authorization mechanism that allows only the token owner to set the market as initialized. Additionally, a check to verify that all the tokens have been sold in “Stage 1” should be implemented.

Status: Resolved

3. Incorrect math leads to underpriced tokens when buying an offer

Severity: Critical

In `programs/gcx/src/instructions/buy_offer.rs:40-53` the token price computation when buying an offer from the marketplace is incorrect.

Specifically, the amount is divided two times by `LAMPORTS_PER_SOL` which leads to an incorrect price due to unit mismatch in the computation.

Consequently, the price denominated in `SOL` required to buy the offer is incorrect and not predictable by users leading to failed transactions and severely underpriced tokens.

Recommendation

We recommend removing both instances of `.checked_div(LAMPORTS_PER_SOL)` from the computation. For reference, we recommend a similar computation in `programs/gcx/src/instructions/initial_buy.rs:30-42` which correctly calculates the token price.

Status: Resolved

4. Token transfers will always fail due to incorrect account seed

Severity: Critical

The `holder_state` account defined in the instruction context in `programs/gcx/src/instructions/transfer_tokens.rs:70-71` is derived via the `b"token_holder"` seed string.

However, since it is derived and initialized via the `b"holder_state"` seed string in all other instances throughout the protocol, the `transfer_token` instruction will always fail since the required `holder_state` PDA does not exist for this specific seed.

Consequently, it is not possible to transfer tokens.

Recommendation

We recommend modifying the seed string from `b"token_holder"` to `b"holder_state"` in the aforementioned location.

Status: Resolved

5. Inconsistent accounting of token decimals throughout the protocol

Severity: Critical

The decimals of `token_amount` are inconsistently accounted for throughout the whole protocol.

The `BuyMarketPlaceOffer` instruction handler, defined in `programs/gcx/src/instructions/buy_offer.rs:16-141`, suggests that the `token_amount` has `0` decimals according to the price calculations in lines `40-54`, which is based on the price per whole token. However, within the same function, the check in line `77` and the token transfer in lines `134-137` require the token_amount to have 9 decimals.

Similarly, the `InitialBuy` instruction handler, defined in `programs/gcx/src/instructions/initial_buy.rs:16-104`, suggests that `token_amount` has `0` decimals according to the price calculations in lines `30-42`, which is based on the price per whole token, and the token transfer in lines `101-104`, which explicitly upscales the amount from `0` to `9` decimals. However, within the same function, the check in line `50` requires the `token_amount` to have 9 decimals in the first place.

Further instances of inconsistent token decimals also exist within `programs/gcx/src/instructions/transfer_tokens.rs`, `programs/gcx/src/instructions/create_offer.rs`, and `programs/gcx/src/instructions/create_token.rs`.

Consequently, this mismatch between decimals could lead to transaction failure, invalid price computation, and wrong token amounts being transferred.

Recommendation

We recommend defining the expected decimals of any token amount and applying them consistently throughout the whole protocol.

Status: Resolved

6. Users can buy tokens from sellers with the “Stage 1” mechanism after the market is initialized

Severity: Major

The `handle_initial_buy` instruction handler, defined in `programs/gcx/src/instructions/initial_buy.rs:16`, allows users to buy tokens from sellers at a fixed price during “Stage 1”.

However, this instruction is still executable by users to buy tokens from the seller in “Stage 2” after the market initialization.

Consequently, this enables a behavior that diverges from the specification and, if the seller rebuys the tokens in the marketplace, could lead to market manipulations.

Recommendation

We recommend applying the behavior described in the specification and disabling the execution of `InitialBuy` instructions after “Stage 1” is completed.

Status: Acknowledged

7. Token holders can create an infinite number of unbacked offers

Severity: Major

In the `handle_create_or_modify_marketplace_offer` instruction, defined in `programs/gcx/src/instructions/create_offer.rs`, the execution checks if the `seller_mint_ata` account holds at least the `amount_offered` tokens.

However, these tokens are not recorded or transferred to any escrow account after the validation.

As a result, the seller can create an infinite number of offers without actually having the required tokens to back them leading to spamming the marketplace with offers that cannot be fulfilled.

Recommendation

We recommend implementing a mechanism to ensure that every offer is backed by the underlying tokens.

Status: Resolved

8. Marketplace offers can be reinitialized

Severity: Major

The `CreateOrModifyMarketplaceOffer` instruction handler defined in `programs/gcx/src/instructions/create_offer.rs` specifies the offer account as `init_if_needed` which instructs Anchor framework to create an account if it did not exist previously or use the currently existing one during execution.

As a consequence, any seller can modify their offer at any point in time without restrictions, which includes, but is not limited to changing its price.

The particularly malicious scenario is as follows:

First, a malicious user sets up an offer. Once a legitimate user wants to accept the offer using `BuyMarketPlaceOffer` instruction, the malicious user can frontrun the legitimate user's transaction and re-initialize the offer with a different price, causing financial loss to the legitimate user.

It is worth noting that currently, frontrunning in Solana requires running a malicious validator, however, a MEV infrastructure project may be available in the future that will facilitate frontrunning attacks.

Additionally, a malicious user might still try to modify the offer in hopes of deceiving legitimate users who would see previous prices in the frontend components.

Recommendation

We recommend disabling the possibility of reinitializing the marketplace offer account by setting it as `init` in the accounts struct.

Status: Resolved

9. Invalid boolean arithmetic leads to a verification bypass

Severity: Major

In `programs/gcx/src/instructions/create_offer.rs`, the `CreateOrModifyMarketplaceOffer` instruction handler implements a verification mechanism in line `21` that is intended to allow offer creation only when the marketplace is initialized and the program is configured to enable market offers.

Specifically, the following boolean arithmetic is responsible for checking for a failing condition and failing the execution if they are not met:

$$\neg program\_state.allow\_market\_offers \wedge \neg mint\_owner\_token\_state.market\_initialized$$

However, this boolean arithmetic is not correct as the logic `AND` makes the program fail execution only if both boolean values are set to `false`.

Considering that anyone can set a market as initialized because "Market can be initialized without seller authorization and before all tokens have been sold", the scenario where the market is initialized (`mint_owner_token_state.market_initialized` is set to `true`) and the program does not allow market offers (`program_state.allow_market_offers` is set to `false`) appears to be easily achieved.

Consequently, marketplace offers can be created at any time by anyone, even if the program's global settings disable it.

Recommendation

We recommend implementing a different guard for creating and updating the offer:

- Creation: We recommend changing the boolean `AND` to `OR`, so that if even a single value required for a successful execution is not set to `true`, the program will fail the execution.
- Update: We recommend not applying the guard so users can still modify or close the offer by setting the `amount` to `0`.

Status: Resolved

10. Assuming negative sign of Pyth price feed exponent can lead to incorrect token pricing

Severity: Major

The token price computations in `programs/gcx/src/instructions/initial_buy.rs:37-41` and `programs/gcx/src/instructions/buy_offer.rs:45-49` are performed on a Pyth price feed which is given by `price` and `expo`, i.e. `price*10^expo`.

However, both instances assume the `expo` to be always negative and therefore reconstruct the price via `price/10^abs(expo)`. Still, according to the first example in the `pyth_sdk_solana::Price` documentation, the `expo` can also be positive.

Consequently, the current implementation can lead to wrong token pricing in case the returned `expo` is not negative.

Recommendation

We recommend implementing the handling of cases where the `expo` value is positive.

Status: Resolved

11. Multiple minting is allowed in the `CreateToken` instruction

Severity: Minor

The `CreateToken` instruction handler in `programs/gcx/src/instructions/create_token.rs:7:38` is responsible for minting initial tokens to an `authority` account and setting the market associated with them as not initialized.

However, since `mint`, `mint_ata`, and `token_state` accounts are marked with `init_if_needed` the Anchor framework will not fail the execution if those accounts are already created allowing the program owner to execute the mint of tokens to the same `authority` multiple times.

This behavior diverges from the specification since it would inflate the amount of the designated initial tokens per user.

We are reporting this issue with minor severity since only the program owner can execute the mint.

Recommendation

We recommend changing the constraints in the `CreateToken` accounts struct so that the `mint`, `mint_ata`, and `token_state` accounts are marked with `init` to prevent subsequent token mints.

Status: Resolved

12. Lack of input validation for the `initialize_program_state` instruction

Severity: Minor

In the `init` function implementation for `ProgramState`, defined in `programs/gcx/src/states/program_state.rs:15:33` there is no validation on the provided data.

Specifically, the parameters `market_owner_fee`, `market_program_fee`, `initial_fee`, and `allow_market_offers` should be within the range of `(0, 100)` as they represent a percentage.

Similarly, `initial_token_price` should be enforced to be greater than `0` to make the subsequent transfer succeed.

Consequently, this can lead to incorrect or undesirable behavior, potentially leading to underflows or division by zero errors.

Recommendation

We recommend adding validation checks to the init function to ensure that `market_owner_fee`, `market_program_fee`, `initial_fee`, and `allow_market_offers` are within the `(0, 100)` range and that `initial_token_price` is greater than `0`.

Status: Resolved

13. Possible precision loss due to incorrect calculations

Severity: Minor

The `BuyMarketPlaceOffer` instruction handler in `programs/gcx/src/instructions/buy_offer.rs:40:54` calculates token prices.

Currently, divisions are intertwined with multiplications. Furthermore, all calculations are done on the `u64` type. Such an approach to the calculation is inherently subject to higher precision losses which may result in an inaccurate price.

Furthermore, the same instruction handler in `programs/gcx/src/instructions/buy_offer.rs:56:70` is calculating the values to be transferred - the actual amount of tokens for sale along with fees for project and market owner. Each value is calculated independently using arithmetic operations which include division. Due to the inherent precision loss associated with integer division it is possible that the sum of all calculated values would not add up to the initial total value. If this scenario occurs, some of the lamports would be stuck in the program account.

Recommendation

We recommend first casting the values to the `u128` type to limit the overflow issues in temporary values. Then, all of the multiplications should be performed. The division should be performed after all other multiplications. Finally, once the calculated value is used in the context of Solana lamports, it should be downcasted to `u64` type.

Furthermore, we recommend calculating the amount of tokens sent to the seller as the difference between the total price and calculated fees. That approach will always use all of the available tokens.

Status: Partially Resolved

14. Missing check allows to set any account as token authority

Severity: Minor

The accounts struct for the `CreateToken` instruction in `programs/gcx/src/instructions/create_token.rs:41:71` contains a member called `authority` that is used to set the `authority` field in the `TokenState` account.

However, the `CreateToken` instruction's accounts struct specifies this member to be `AccountInfo` causing the instruction to accept any account provided, making it possible to use an incorrect one.

We are reporting this issue with minor severity since only the program owner can execute the mint.

Recommendation

We recommend setting the `authority` as a `Signer` in the `CreateToken` instruction's account to prevent accidentally setting an incorrect value in the `TokenState` account.

Status: Resolved

15. Program state initialization is fully permissionless

Severity: Minor

The `InitializeState` instruction, defined in `programs/gcx/src/instructions/program_state.rs`, allows anyone to permissionlessly create their own `program_state` PDA. This allows users to own and operate their instance of the protocol, while an indefinite number of such instances can coexist, all using the same program ID.

Although this permissionless nature might be intended, it also enables scammers to trick users into interacting (e.g. initial buy or marketplace buy/sell) with a malicious `program_state` without any suspicion, due to interacting with a known & trusted program ID.

Recommendation

We recommend evaluating the risks of the permissionless nature of the program state initialization. Alternatively, a whitelist of allowed accounts could be implemented.

Status: Resolved

16. Holder status is not used

Severity: Informational

The `HolderState` account, defined in `programs/gcx/src/states/holder_state.rs`, specifies a member of `boolean` type called `active`.

However, it has been observed that this value is not used in the program and it is always set to `true` when the account is initialized and there are no paths in the code to update it.

Recommendation

We recommend implementing a security measure using the `active` parameter. A granular approach to user state should be implemented that clearly defines scenarios where a particular `HolderState` is considered active. Then, this parameter should be checked when executing an instruction that is using a specific `HolderState`, so that it can only be completed on active states. For instance, `CreateOrModifyMarketplaceOffer` instruction could benefit from such a security measure.

Alternatively, if such a security measure would be deemed unnecessary after analyzing the business reasons and protocol's design, then `active` could be removed from `HolderState` data structure as optimization.

Status: Resolved

17. Miscellaneous comments

Severity: Informational

Miscellaneous recommendations can be found below.

Recommendation

The following are some recommendations to improve the overall code quality and readability:

- The `Error` variant returned in `programs/gcx/src/instructions/create_offer.rs:22` covers different failing conditions. Ideally, two separate `Error` variants should be created and the verification mechanism should be split into two separate `if` statements.
- The fees within the protocol can only be set as whole number percentages. Changing them to be reflected as "basis points" would allow fractional fees and introduce more flexibility to the protocol.
- The protocol uses the native system instruction to transfer native tokens. If such a transfer would cause the sender or receiver to not have enough lamports to be rent-exempt, then the transaction would fail causing a subtle DoS condition. Additionally, native SOL transfers are not indexed well by blockchain explorers. Wrapped SOL should be used instead.
- There are numerous `unwrap` calls throughout the codebase. Calling an `unwrap` function without proper error handling is considered a bad practice. At the very least, `expect` function should be used instead to provide a custom error message.
- The `ProgramState` struct in `programs/gcx/src/states/program_state.rs` defines various setter functions. Most of them are not used anywhere in the codebase, namely: `set_authority`, `set_initial_usd_price`, `set_transfer_initial_fee`, `set_owner_fee_percentage`, `set_program_fee_percentage`. Unused functions should be removed to increase the code readability.

- The `TransferTokens` Accounts struct assumes the `holder_state` account to be already initialized. However, the instruction handler in `programs/gcx/src/instructions/transfer_tokens.rs` contains an `if` statement that checks if it is initialized and sets its data in case it is not. This `if` statement is redundant and can be deleted.
- The `mint_key` variable in `programs/gcx/src/instructions/buy_offer.rs:127` is unused. This account is necessary only for constraints in other accounts and there is no need to assign it to a variable.
- The `GCCError` enum variant defined in `programs/gcx/src/instructions/buy_offer.rs:201` contains a typo. It should be `ArithmeticError` while it currently is `ArithmaticError`.
- The `BuyMarketPlaceOffer` instruction handler in `programs/gcx/src/instructions/buy_offer.rs:140` is modifying the `MarketplaceOfferState` account by subtracting the bought tokens. It is worth considering closing the account once the `amount_offered` value reaches `0` for a given offer. That way, less data will be kept on blockchain and some lamports could be freed.
- The Anchor version in use has dependencies that are affected by vulnerabilities such as `ed25519-dalek` which is affected by RUSTSEC-2022-0093. We recommend updating Anchor to the latest version.
- The Anchor version in use has dependencies that are affected by vulnerabilities such as `curve25519-dalek` which is affected by RUSTSEC-2024-0344. We recommend updating Anchor to the latest version.
- The value of the `STALENESS_THRESHOLD` defined in `programs/gcx/src/instructions/buy_offer.rs:14` and `programs/gcx/src/instructions/initial_buy.rs:14` is only set for testing purposes and should not be used in production.
- The `market_place_offer` account definition in the instruction context in `programs/gcx/src/instructions/buy_offer.rs:158` is missing the `pub` keyword.
- The token decimals are hard coded as number 9 in `programs/gcx/src/instructions/create_token.rs:45`, `programs/gcx/src/instructions/initial_buy.rs:103`, and `programs/gcx/src/instructions/transfer_tokens.rs:50`. We recommend defining a constant that is used throughout the whole protocol.
- The `mint::authority` of the `mint` account defined in the instruction context in `programs/gcx/src/instructions/create_token.rs:42-50` is set to itself, i.e. `mint::authority = mint`. We recommend reevaluating if this self-authorizing behavior is ultimately intended.

Status: Partially Resolved