The Graph A-1

Security Audit

February 24th, 2023
Version 1.0.0

Presented by 0xMacro

Table of Contents

- Introduction
- Overall Assessment
- Specification
- Source Code
- Issue Descriptions and Recommendations
- Security Levels Reference
- Disclaimer

Introduction

This document includes the results of the security audit for The Graph's smart contract code as found in the section titled ‘Source Code’. The security audit was performed by the Macro security team from February 13 to February 14, 2023. Review of fixes was conducted on February 20, 2023.

The purpose of this audit is to review the source code of certain The Graph Solidity contracts, and provide feedback on the design, architecture, and quality of the source code with an emphasis on validating the correctness and security of the software in its entirety.

Disclaimer: While Macro’s review is comprehensive and has surfaced some changes that should be made to the source code, this audit should not solely be relied upon for security, as no single audit is guaranteed to catch all possible bugs.

Overall Assessment

The following is an aggregation of issues found by the Macro Audit team:

Severity	Count	Acknowledged	Won't Do	Addressed
High	3	-	-	3
Medium	1	-	-	1
Low	3	-	1	2
Code Quality	4	-	-	4
Informational	1	-	-	-
Gas Optimization	4	-	-	4

The Graph was quick to respond to these issues.

Specification

Our understanding of the specification was based on the following sources:

- An information summary delivered over chat.
- Discussions on Slack with the The Graph team.

Trust Model, Assumptions, and Accepted Risks (TMAAR)

Entities

- **Gateway** – The nodes that process queries
- **User** – Creates a Subscription in the contract, funded by themselves or by someone else.
- **Subscription** – A struct with a start date, end date, and rate per second. Off-chain, intended to represent a subscription to a gateway's processing resources.
- **Pending Subscription** – A struct representing a potential subscription, to be fulfilled by a 3rd party (specifically, Banxa for the initial MVP).

Trust Model

- Primary goal is to make payments between Users and Gateways more trustless.
- A User should be able to unsubscribe and get a refund of their balance for any unused Subscription time.
- It is the gateway's sole discretion, off-chain, whether or not a subscription is valid. Users are expected to use rates that the gateway agrees with off-chain.

General Assumptions

- Subscription statuses are intended to be aggregated and queried through The Graph subgraphs.

Source Code

The following source code was reviewed during the audit:

- **Repository:** subscription-payments
- **Commit Hash (initial):** `81a44aede886fc972b2f2422038a804f00789966`
- **Commit Hash (final):** `71a8bff74ef30f1e530ae9c922dcfc5e7a2c7983`

Specifically, we audited the following contracts within this repository:

Contract	SHA256
contracts/Subscriptions.sol	`11c36985938d7c3e8c9265e8c3d32444dd57cdb09f82529ce0b76d776f1408c5`

Note: This document contains an audit solely of the Solidity contracts listed above. Specifically, the audit pertains only to the contracts themselves, and does not pertain to any other programs or scripts, including deployment scripts.

Issue Descriptions and Recommendations

Click on an issue to jump to it, or scroll down to see them all.

~~H-1~~	`subscribe()` not callable for other user after their first subscription expires
~~H-2~~	`extendSubscription()` front-run vulnerability
~~H-3~~	`fulfil()` reverts or loses funds when user has an active subscription
~~M-1~~	Calling `subscribe()` with incorrect values can lock funds in contract
~~L-1~~	`subscribe()` griefing attack
L-2	Unsafe ERC-20 transfer
~~L-3~~	Minor Reentrancy
~~Q-1~~	`locked()` may produce inaccurate result due to cast truncation
~~Q-2~~	Additional arguments for `Init` event
~~Q-3~~	Inaccurate comment
~~Q-4~~	Underflow revert
~~G-1~~	Unnecessary `require()`
~~G-2~~	Duplicate computation
~~G-3~~	Unnecessary repeat storage reads & writes in `collect()`
~~G-4~~	Duplicate logic in `unsubscribe()`
I-1	Mistakes with user-controlled rates

Security Level Reference

We quantify issues in three parts:

1. The high/medium/low/spec-breaking **impact** of the issue:
    - How bad things can get (for a vulnerability)
    - The significance of an improvement (for a code quality issue)
    - The amount of gas saved (for a gas optimization)
2. The high/medium/low **likelihood** of the issue:
    - How likely is the issue to occur (for a vulnerability)
3. The overall critical/high/medium/low **severity** of the issue.

This third part – the severity level – is a summary of how much consideration the client should give to fixing the issue. We assign severity according to the table of guidelines below:

Severity	Description
(C-x) Critical	We recommend the client **must** fix the issue, no matter what, because not fixing would mean **significant funds/assets WILL be lost.**
(H-x) High	We recommend the client **must** address the issue, no matter what, because not fixing would be very bad, *or* some funds/assets will be lost, *or* the code's behavior is against the provided spec.
(M-x) Medium	We recommend the client to **seriously consider** fixing the issue, as the implications of not fixing the issue are severe enough to impact the project significantly, albiet not in an existential manner.
(L-x) Low	The risk is small, unlikely, or may not relevant to the project in a meaningful way. Whether or not the project wants to develop a fix is up to the goals and needs of the project.
(Q-x) Code Quality	The issue identified does not pose any obvious risk, but fixing could improve overall code quality, on-chain composability, developer ergonomics, or even certain aspects of protocol design.
(I-x) Informational	Warnings and things to keep in mind when operating the protocol. No immediate action required.
(G-x) Gas Optimizations	The presented optimization suggestion would save an amount of gas significant enough, in our opinion, to be worth the development cost of implementing it.

Issue Details

~~H-1~~ `subscribe()` not callable for other user after their first subscription expires

TOPIC	Use Cases	
STATUS	Fixed	
IMPACT	Spec Breaking	`subscribe()` should be callable for other users with expired subscriptions.
LIKELIHOOD	N/A	

The `subscribe()` function mistakenly compares the `user` subscription end date against `block.number` instead of `block.timestamp`. This prevents a user X from calling `subscribe()` for a user Y if user Y has an expired subscription.

Consider updating `block.number` to `block.timestamp`.

~~H-2~~ `extendSubscription()` front-run vulnerability

TOPIC	Front-Running	
STATUS	Fixed	
IMPACT	Critical	Funds are stolen.

LIKELIHOOD	Medium	Increases linearly with over-approval amounts.

In `extendSubscription()`, the user's current subscription is used to calculate how many additional tokens are required to extend the subscription up to the specified `end`.

However, a user may be able to get the extender to pay more than they intend. For example:

1. User X has a subscription from time `200` to `500` with rate `7`. Current time is `230`.
2. Extender calls `extendSubscription()` with user X's address and an end time of `800`.
3. User X front-runs transaction with:
   - `unsubscribe()`, also getting a refund
   - `subscribe()` with `start` as now, with `end` as a small amount after (such as `315`), and with `rate` as high as they can afford.
4. Extender, intending to pay for `250` seconds of time, is now paying for `435` seconds, and at a much higher rate than intended.

This issue affects extenders linearly with approval size.

Consider updating `extendSubscription`'s parameters to include a `intendedRate` so the extender's paid rate cannot be front-run.

Also consider replacing `end` with `secondsToExtend` so extenders cannot pay for more time than they intend.

~~H-3~~ `fulfil()` reverts or loses funds when user has an active subscription

TOPIC	Edge Case	
STATUS	Fixed	
IMPACT	High	Common case: Use case is blocked. Worst case: User loses funds from their current subscription, by malicious intent or by grief.
LIKELIHOOD	Medium	User must have an active subscription, but can be griefed into having one. Third party will likely not have a subscription, and can be griefed into having one.

A use case supported by the Subscriptions contract is allowing a third party to call `fulfil()` on a user's pending subscription, converting the pending into an actual subscription and paying for it.

If the targeted user already has an existing subscription, `unsubscribe()` is called to refund the user before converting the pending subscription.

However, the `unsubscribe()` function only refunds `msg.sender`, which is different than the user in this use case. This causes one of the following issues:

- If the `msg.sender` has an active subscription, the `msg.sender`'s subscription is unsubscribed, and the target user's subscription is overwritten, losing access to those funds.
- If the `msg.sender` does not have an active subscription, the call reverts with `no active subscription`, and the use case is blocked from functioning.

Note that the subscription status of either party can be freely switched from no-subscription to active-subscription by way of `subscribe()`, allowing a griefer to set up these scenarios at low cost.

Consider implementing an internal function `_unsubscribe(address)` which accepts an address, and calling it from `unsubscribe()` with `msg.sender`, and from `_subscribe()` with `user`.

~~M-1~~ Calling `subscribe()` with incorrect values can lock funds in contract

TOPIC	Input Ranges	
STATUS	Fixed	
IMPACT	High	`user` loses funds.
LIKELIHOOD	Low	User would need to call `subscribe()` with `start` and `end` parameters orders of magnitude larger than expected.

The `subscribe()` function allows `start` and `end` values to have a maximum value of `type(uint64).max`, but the `start` value can be truncated during cast to `int64` on lines 361-362 of `unlocked()`:

```
uint256 len = uint256(
    SignedMath.max(
        0,
        **int256(int64(_subEnd)) -
            int256(Math.max(block.timestamp, _subStart))**
    )
);
```

As a result, `unsubscribe()` can mis-calculate the refund and leave a portion of funds locked in the contract.

Consider restricting subscription `start` and `end` to a maximum value of `type(int64).max`.

~~L-1~~ `subscribe()` griefing attack

TOPIC	Open Access	
STATUS	Fixed	
IMPACT	Medium	Blocks an uncommon use case.
LIKELIHOOD	Low	No one benefits from the attack.

When user X calls `subscribe()` to create a subscription for user Y, a malicious user Z can front-run the transaction by calling `subscribe()` themselves with a long `end` and a minimal `rate`.

This can disrupt the [assumingly less-common] use case of a machine user granting a user a subscription upon some event.

Consider one or many of:

1. Extending the pending subscription feature to cover this use case
2. Documenting the limits of this approach, particularly not to build a system around it
3. Removing this feature

L-2 Unsafe ERC-20 transfer

TOPIC	Ecosystem	
STATUS	Wont Do	
IMPACT	Medium	Blocks certain tokens from being used as currency.
LIKELIHOOD	Low	The number of tokens that fall under this category may be limited.

Not all ERC-20 tokens return a boolean for their transfer functions. These tokens are not compatible with the Subscriptions contract as currently written, since it requires that they return `true`.

Consider using Open Zeppelin's SafeERC20 helper function if support for these tokens is desired.

RESPONSE BY THE GRAPH

> Won't fix. Only applies to nonconforming ERC-20 implementations that don't return boolean values from `tranfer()`. We don't intend to support these implementations.

~~L-3~~ Minor Reentrancy

TOPIC	Re-entrancy	
STATUS	Addressed	
IMPACT	Low	Locally, exploiter only loses money. Might create issues for future integrations, though unlikely.
LIKELIHOOD	Low	Only applicable if a future integration somehow introduces a vulnerability.

The `_subscribe` has a minor reentrancy vulnerability:

- If the user has a sub, then `_unsubscribe` is called
- This transfers tokens to the user (for a refund), giving an opportunity for reentrancy
- This reentrancy occurs *before* `subscriptions[user]` and the new epoch states get updated.

However, this causes the attacker to lose money, as their old subscription data is overwritten with the new, with no chance to retrieve the funds for the old.

This shouldn't cause any problems. However, if extra safety is desired, consider adding OZ's `nonReentrant` modifier to `subscribe`, `unsubscribe`, and `fulfil`.

RESPONSE BY THE GRAPH

> We've added a comment explaining the potential vulnerability.

~~Q-1~~ `locked()` may produce inaccurate result due to cast truncation

TOPIC	Data Views
STATUS	Fixed
QUALITY IMPACT	High

Similar to M-2, `locked()` also may cause cast truncation and yield incorrect values. See line line 332:

```
uint256 len = uint256(
    SignedMath.max(
        0,
        **int256(Math.min(block.timestamp, _subEnd)) - int64(_subStart)**
    )
);
```

This function is not utilized internally so no vulnerability is created within this contract.

Consider restricting subscription `start` and `end` to a maximum value of `type(int64).max`.

~~Q-2~~ Additional arguments for `Init` event

TOPIC	Events
STATUS	Fixed
QUALITY IMPACT	Medium

The values for `epochSeconds` and `uncollectedEpoch` may be beneficial for off-chain consumers. Consider adding these to the `Init` event.

~~Q-3~~ Inaccurate comment

TOPIC	Documentation
STATUS	Fixed

QUALITY IMPACT	Low

The following comment is inaccurate:

```
/// @dev Second param required, but currently unused.
```

Consider updating the comment accordingly.

~~Q-4~~ Underflow revert

TOPIC	User Experience
STATUS	Fixed
QUALITY IMPACT	Low

Calling `fulfil()` is called after the pending subscription's `end` timestamp causes the transaction to revert. No harm is done in this scenario except perhaps to user/developer experience.

Consider adding a validation check against the underflow to provide a useful error message. Note that this is likely uncommon case.

~~G-1~~ Unnecessary `require()`

TOPIC	Redundant Code

STATUS	Fixed
GAS SAVINGS	Low

`msg.sender` can never be zero in `unsubscribe()`. Consider removing this check.

~~G-2~~ Duplicate computation

TOPIC	Redundant Code
STATUS	Fixed
GAS SAVINGS	Low

In `unsubscribe()`, `minAmount` is assigned to the same value as `amountUsed`. Consider consolidating them into one variable.

~~G-3~~ Unnecessary repeat storage reads & writes in `collect()`

TOPIC	Storage I/O
STATUS	Fixed
GAS SAVINGS	Medium

Repeated reads and writes to the same storage is discounted, but still expensive. Consider assigning `uncollectedEpoch` to a new local variable for use within the `collect()` loop, and then writing its value back to `uncollectedEpoch` after the loop completes.

This reduces `n` storage writes to 1 storage write, saving gas.

~~G-4~~ Duplicate logic in `unsubscribe()`

TOPIC	Redundant Code
STATUS	Fixed
GAS SAVINGS	Low

In `unsubscribe()`:

- The subscription is required to be not expired (`sub.end > _now`)
- Therefore, the `if` / `else if` can be simplified to an `if` / `else`
- Then, the `setEpochs(sub.start, sub.end, -int128(sub.rate));` can be moved out and above the `if` / `else` statements, as it should occur in both cases.

I-1 Mistakes with user-controlled rates

TOPIC	Input Validation
IMPACT	Informational ✳

As described in TMAAR, the responsibility of setting the correct subscription rate is solely on the user. When working with this smart contract:

- Users should be wary of mistakes in setting a subscription rate, as an incorrect rate can cause loss of funds to the user.
- Gateways should be wary when upgrading their off-chain rate validation, and consider backwards compatibility for existing subscriptions.

Disclaimer