GOATAI

Documentation for Security Audit Request

Summary

Name:
GOATAI (GOAT Athletics Inc.)

Contract:
GOATAI_ERC20

Contract Types:
- ERC20 Governance Token with Fee-on-Transfer
- GovernorUpgradeable
- TimelockController

Token Decimals:
18

Total Supply:
1,000,000,000,000 tokens (1 trillion)

Solidity Version:
0.8.28

Blockchain Network on which contracts will be deployed:
Base chain - Ethereum Layer 2.

Upgradeability:
Immutable, non-upgradeable

License:
MIT

GitHub Repository:
https://github.com/GOAT-Athletics-Inc/contracts

Security Contact:
dev@goatathletics.ai

Project Overview

Project Description

GOAT Athletics Inc, from here on called GOATAI, is a charity, sports and meme governance token. Its mission is to raise funds to power the charitable initiatives of the Global Running Foundation to drive positive change in the areas of health, sustainability and opportunity. Funds will be raised from buy and sell trading fees.

The GOATAI token is an ERC20 token, and is represented by the GOATAI_ERC20 contract, file: ***contracts/GOATAI_ERC20.sol***.

The GOATAI_ERC20 contract extends audited and battle-tested OpenZeppelin contract templates where possible, with the exception of the fee-on-transfer logic.

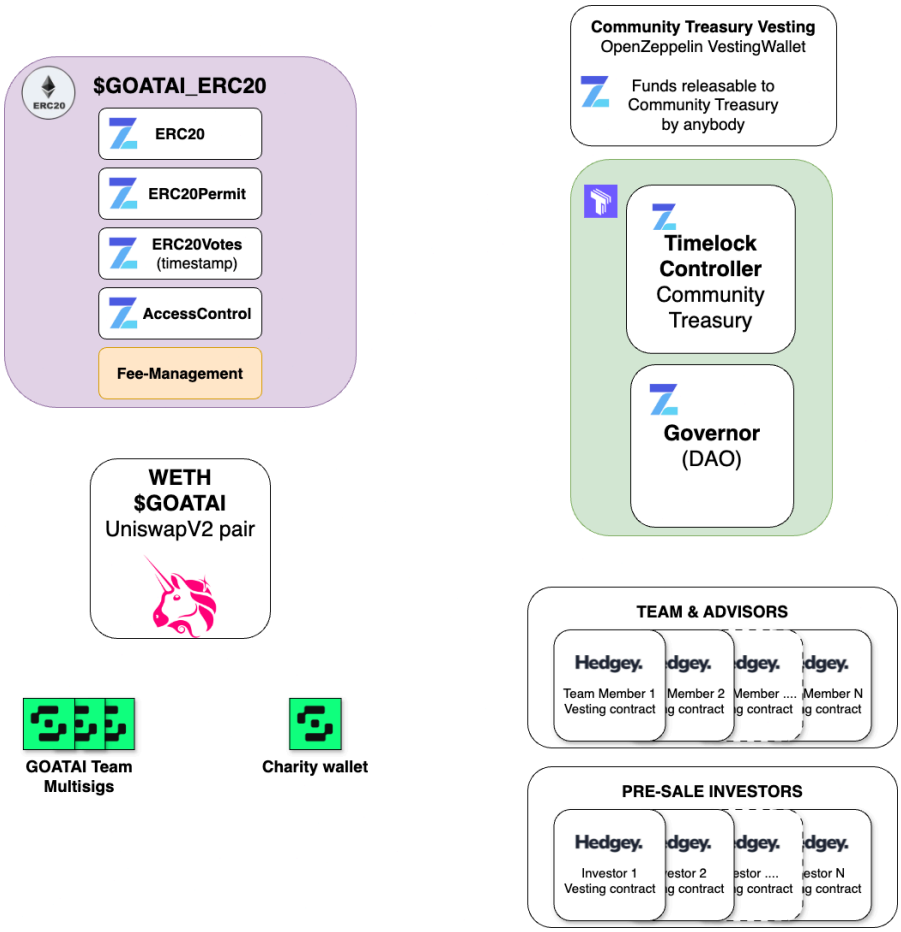
Core Features

1. **Token Operations**
    - Standard ERC20 transfers
    - Fee-collection mechanism for buy/sell trades
    - Voting power tracking and delegation
    - Permit functionality for gasless approvals

2. **Fee Management**
    - Configurable buy/sell fees
    - Multiple fee recipients
    - Automatic fee distribution
    - Partial or full fee burning

3. **Timelocked Governance (outside of audit scope)**
    - Proposal creation
    - Voting & delegated voting on proposal. Options: for / against / abstain.
    - Time-locked delays between proposal, voting and execution
    - Cancelling proposal (by admin or proposer)

Architecture



Core Components

1. **GOATAI Token Contract**
    - Implements ERC20 functionality
    - Controls access through role-based system
    - Handles fee collection and distribution
    - Manages voting capabilities
    - Delegates votes to transfer recipient if no delegates are set

Modules used: OpenZeppelin Contracts v5.2.0

- ERC20
- ERC20Permit
- ERC20Votes
- Nonces
- AccessControl
- VestingWallet - Community Treasury Vesting Smart Contract

External Integrations

Tally.xyz

Tally.xyz will be used to deploy:

- Governor contract (OpenZeppelin)
- TimelockController contract (OpenZeppelin)

Hedgey Finance

Deployment of Vesting Contracts, allowing for recipients to participate in on-chain governance.

UniswapV2

On Uniswap:

- the liquidity pair is created (UniswapV2 contracts will be selected, in order to support fee-on-transfer)
- Trading liquidity is added
- Trading activity will occur

Fee-On-Transfer Mechanism

GOATAI_ERC20 has a fee-on-transfer mechanism. Fees are collected as a portion of every buy and sell trade.

Fee

Although the contract will be initialized with the same buy and sell fees, GOATAI_ERC20 supports different values for buy and sell fees.

The fees will be collected in GOATAI tokens, and are calculated as:

Buy_fee_amount = buy_amount x buy_fee_portion
Sell_fee_amount = sell_amount x sell_fee_percentage

Note: The portion is actually stored in basis points, so as to allow for 2 decimal points percentage precision. E.g. if the fee is 3.5%, then the fee will be stored in the contract as *350*.

Recipients and Fee-Splits

The **fee recipients** are the **same for buy and sell** trades. When changing a recipient address and/or split, the effect will be applied for all non-exempt trades.

At the point of fee collection, the contract will distribute the fee to the recipients according to the recipient “splits” (an allocation of the fee). The remainder of the fee that has not been allocated to recipients gets burned:

- If no recipients are set, all of the fee will be burned.
- If the sum of the recipient splits is 80%, then 20% of the fee will be burned.
- If the recipients’ splits sum up to 100%, none of the fee will be burned.

These are hard-coded constraints that, combined with the contract’s immutability will give the following guarantees:

- Buy and sell fees cannot be higher than 7.5%. This will make it such that nobody, neither the team, nor any hacker, and not even the DAO, can abuse the fee mechanism and set super high fees on trades.
- The number of recipients cannot be higher than 5. This is to guarantee that the fee structure will not get too complex, nor end up with too many recipients (hundreds/thousands) and too high gas.

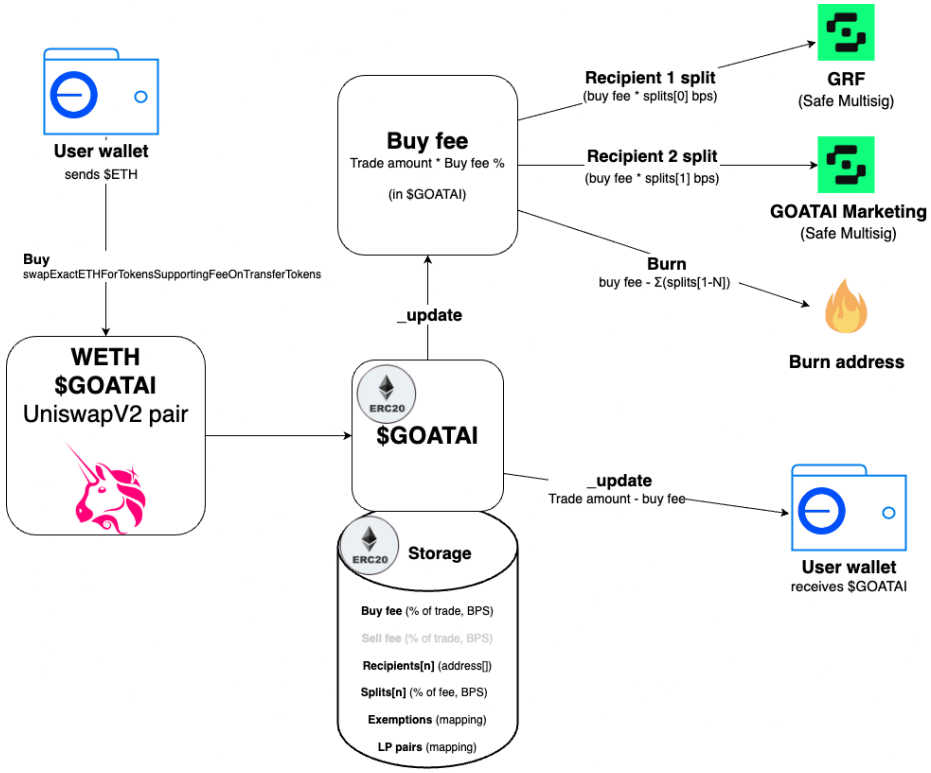
Exemptions

The contract allows for some accounts to be **exempt** from fees. This is intended to be for the purpose of charity wallets, the DAO/Community Treasury and Operations/Marketing multi-sig’s.

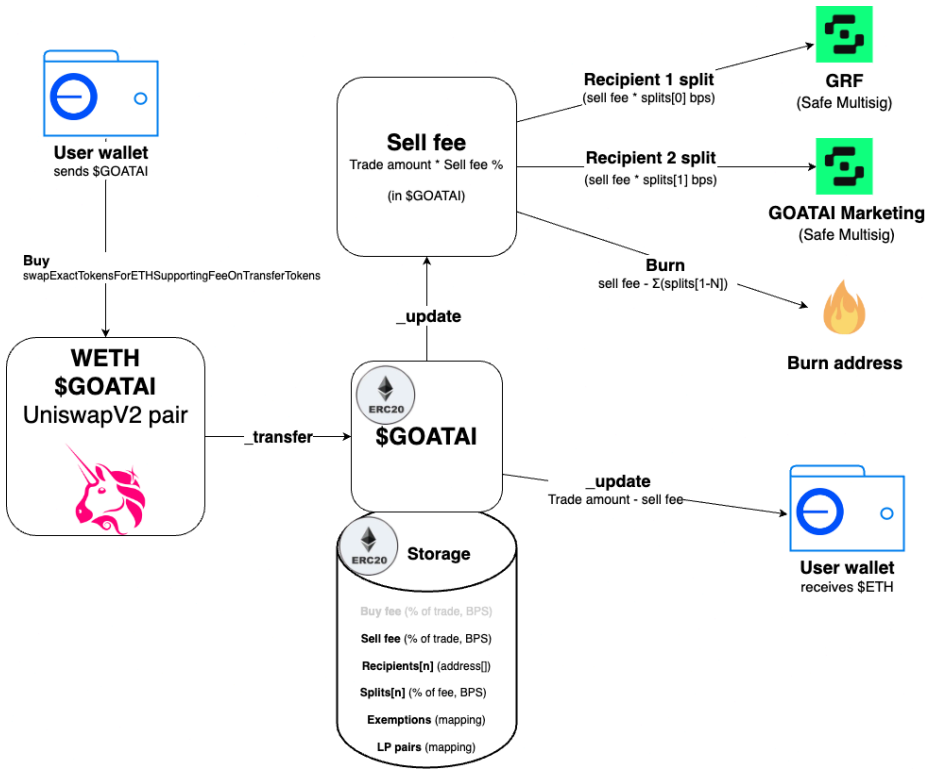
LP’s

To distinguish between regular transfers and trades, Liquidity Pair (LP) addresses will be stored as a mapping in the contract storage. Only 1 liquidity pair is foreseen for the foreseeable future, but a mapping allows for further ones if needed.

Fee-on-transfer for buy trades



Fee-on-transfer for sell trades



Governance

ERC20Votes

GOATAI_ERC20 extends OpenZeppelin's **ERC20Votes** contract. The ERC20 token's clock will be in timestamp format.

In the ***_update*** function, we implement a logic for the token recipient to automatically **self-delegate** votes, if no delegates are set for that account. See section on Improving Voting UX at the expense of gas-efficiency.

Governor + Timelock

The Governance of this token is managed via OpenZeppelin's **GovernorUpgradeable** contract with a **TimelockController**, deployed through Tally.xyz.

OpenZeppelin's GovernorUpgradeable and TimelockController contracts have already been audited, and the code in those contracts will not be in the scope of this security audit, but included in architecture overview.

Having a combined voting delay and voting period greater than 0 should prevent flash loan attacks on the Governance votes to access the Community Treasury.

Parameters used in configuration*:

- **Proposal threshold** - 3%**.
- **Quorum -** ~5%**
- **Voting delay** - 3 days**, to allow voters to buy tokens to vote. This also allows some time to review the proposal, in case of malicious proposals. We might
- **Voting period** - 5 days**, to ensure enough time for voters to participate.
- **Timelock delay -** 3 days**

*Numbers can change over time, to better fit the community as it grows and evolves.

**The team reserves the right to change its mind on this parameter in between the time of submission of this document and the time when the Governor + Timelock are deployed via Tally.

Vesting

GOATAI tokens will be subject to vesting periods for these entities:

- Community Treasury
- Founding Team members & Advisors
- Pre-sale investors
- Team Multisig Accounts:
    - Operations & Development
    - Marketing

All Vesting mechanisms will use battle-tested and audited smart contracts, and will therefore be out of audit scope but included in architecture overview.

Community Treasury

The Community Treasury vesting contract will be managed with OpenZeppelin's **VestingWallet** contract. The reason for this choice is for the simplicity of the contract, and the ability for anybody to release the funds to the DAO without requiring a DAO vote for the funds to be received.

Multi-sigs, Team and Pre-sale Investors

All other vesting mechanisms will be handled via **Hedgey Finance** smart contracts. The reason for this choice is that Hedgey's vesting smart contracts allow the recipients of these contracts to have voting power before the tokens have been unlocked, via delegation of the voting power from the smart contract to the recipient.

Roles and Permissions

GOATAI_ERC20's permission model is based on OpenZeppelin's AccessControl. Here is a breakdown of the roles and their permissions.

FEE_MANAGER_ROLE
Fee managers will be capable of setting:

- Buy fee - as a portion of the total trade
- Sell fee - as a portion of the total trade
- Recipient addresses & number of recipients
- Fee Split for each of the recipients
- Burn amount - implicitly - as a remainder of the fee splits for all other recipients

This role will be initially taken by the team's multisig and will be eventually moved to the DAO.

DEFAULT_ADMIN_ROLE
Admins will be capable of:

- Granting and revoking other accounts the role of *DEFAULT_ADMIN_ROLE*
- Granting and revoking other accounts the role of *FEE_MANAGER_ROLE*
- Add/remove an address to the mapping of LiquidityPair addresses
- Add/remove an address to the mapping of fee-exempt addresses

This role will be initially taken by the team's multisig and will be eventually moved to the DAO.

Initial Setup Admin
In addition to these, there is a temporary role - not explicitly coded in the ERC20 contract - i.e. the role of the initial setup admin. This role will only exist on the day of the Token Generation Event (TGE), and shall perform all of the following duties on that day.

- Receive the total supply of the token on contract creation.
- Create vesting contracts for each recipient (team member/investor) and team multisig (operations, development) and for the DAO, according to vesting schedule.
- Transfer the locked tokens, subject to vesting, to the vesting contracts.
- Transfer the unlocked tokens to the multi-sig accounts and to the DAO as per tokenomics.
- Create the liquidity pair for the token
- Add liquidity for the token

Tokenomics

- **Max Total Supply:**
  - 1,000,000,000,000 tokens (1 trillion)
  - No further minting is possible
  - Deflationary mechanism with fee-on-transfer mechanism

- **Fee-on-transfer & Burning:**
  - Buy/Sell fee
    - initially set at 6% buy and sell, and split into 70% charity, 25% operations, 5% burn.
    - Fee % and structure can be changed based on community feedback and voting, and will be fully controlled by the DAO when it will assume the FEE_MANAGER_ROLE.
  - Exemptions for certain addresses (e.g. charity, dao, operations/marketing)

- **Staking:**
  - N/A

- **Airdrops:**
  - Originating from “Community Incentives” token allocation (operated by multi-sig)

Token Distribution & Release schedule

Lidiquity for Launch on DEX	100% Unlocked at TGE
Community Incentives & Rewards	100% Unlocked at TGE Released gradually over time
Community Treasury	Locked for 6 months. Then linearly over 2.5 years
Operations & Development	25% unlocked at TGE. Then linearly over 3 years
Marketing & Partnerships	25% unlocked at TGE. Then linearly over 3 years
Private Sale	Locked for 1 year. Then linearly over 1.5 years
Team & Advisors	Locked for 1 Year. Then linearly over 3 years

Considerations

Invariants

1. Total supply only changes through minting and burning
2. Fee percentages never exceed MAX_FEE_BPS
3. Fee recipient splits never exceed 100%
4. Number of recipients never exceeds MAX_NUM_RECIPIENTS

Known Assumptions & Risks

- **Potential Attack Vectors:**
    - Flash loan attacks affecting governance votes should be prevented by having time-delays between proposal and voting and having a voting period.
    - Centralization of votes for large token holders is possible.
    - Fee bypass mechanisms (ensure no whitelist exploits)
- **Fee mechanism**
    - Abusive fee amount is mitigated by a maximum fee cap of 7.5%.
    - Potential small rounding errors in fee-splits calculations, leading to dust amounts. Mitigation: dust is burned, so as to not give advantages to any other recipient.
    - Potential malicious recipient address
- **Access Control**
    - Admin key security. As a mitigation, the admin account will be a Safe multisig with 3/5 signatories initially. We will increment to 4/6 signatories when the team is bigger.

Improving Voting UX at the expense of gas-efficiency

In the usual implementation of ERC20Votes, when an address receives tokens, it only assumes voting power if and when it delegates to itself. The self-delegation is usually an explicit separate step in order to save on gas for regular transfers and trades.

The problem with this is that, although gas is saved, the Voting UX is confusing for 1st-time participants. Most average users are not aware about this implementation, and many would likely either be unaware or forget to self-delegate before the voting checkpoint.

GOATAI is a governance token at its core. We want users to be able to vote with the tokens they bought or received via airdrop/transfer, with no need to remember self-delegating before voting, nor understanding the ERC20Votes implementation.

We made a decision to implement an automatic self-delegation logic in the ***_update*** function if no delegates are set for that account. This increases the gas consumption for:

- 1st-time transfers to or from an account.

    a. Checks if the recipient address is not an LP
    b. Checks if the account has already delegated votes to itself or others
    c. If no delegates are set, then the recipient's votes are delegated to itself.
- All successive transfers (checks for a. and b. are still performed).

Automated tests written in Forge in (*TradeTestsSimple.t.sol - test_TradeSimple_DelegateGasTest)*, were run both with and without the self-delegation code, and the increased gas consumption was logged as follows:

- 1st time transfer: 71,705 gas units (106,156 gas with self-delegation, 34,406 without)
- 2nd time transfer: 7,624 gas units (14,134 gas with self-delegation, 6,510 without)

We used the historical data for the daily average gas price on the Base chain (source: Basescan, csv here), and compared the prices in wei, with the ether price on each day (source: Basescan, csv here), in order to retrieve the historical gas price in USD for each day.

We have restricted the range to count data points from after the Dencun upgrade on March 13, 2024, which significantly improved gas efficiency for Layer 2 chains until 17 February 2025.

Average gas price	**Median gas price**	**Peak gas price**
$0.000000306 / gas unit	$0.000000221 / gas unit	$0.000002051 / gas unit

We could then estimate the following **additional** gas cost in USD terms for the self-delegation logic.

	Average	**Median**	**Peak**
1st time transfer	$0.0220	$0.0158	$0.1472
2nd time transfer	$0.0023	$0.0017	$0.0156

Based on these numbers, and the benefit that the voting experience gets with the self-delegation logic, we deemed the additional gas worth it.

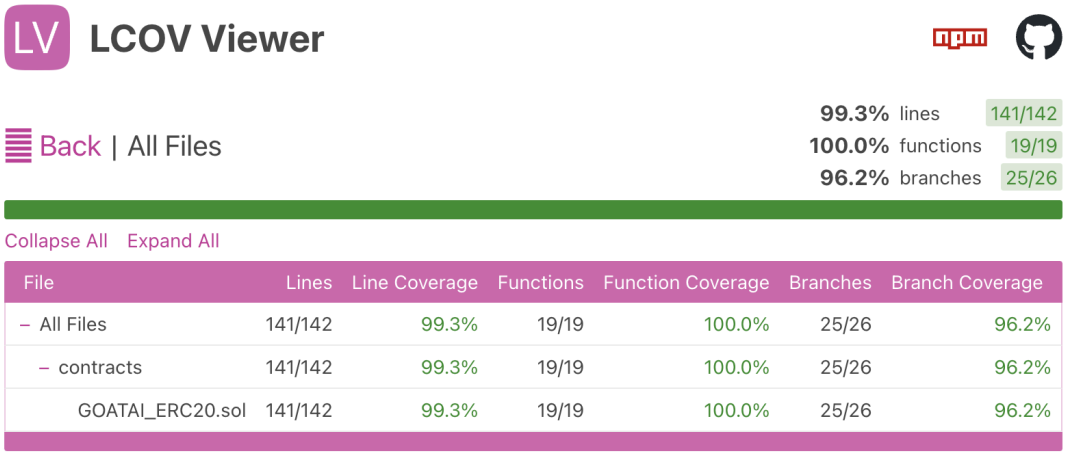
Testing and Coverage

Checks performed

✅ Ran **Forge unit-tests & forge coverage:**

```
forge coverage \
  --report lcov \
  --report summary \--no-match-coverage "(script|test)"
```

Lines	Statements	Branches	Functions
99.30% (141/142)	99.37% (158/159)	96.15% (25/26)	100.00% (19/19)



✅ Ran **Slither** (static analysis)

Results (excluding issues relating to OpenZeppelin modules, and informational issues):

- **(Medium-level) _calculateFee** performs a multiplication on the result of a division
  - The dust produced is really minimal (a couple dozen tokens out of trade of billions of tokens) and gets burned.
  - The added verbosity/complexity of refactoring this does not seem worth it.

✅ Ran **Echidna** (fuzz testing)

Fuzz tested for these Invariants.

```
forge build &&
echidna test/GOATAI/EchidnaGOATAI.sol --contract EchidnaGOATAI_ERC20
```

Deployment Plan

1. Safe Multisig accounts creation
    - Operations & Development (aka “Initial Admin”)
    - Marketing
    - Community Rewards

2. Contracts creation
    - Deploy **GOATAI_ERC20** contract via deployment script from local desktop machine.
        - 1 trillion tokens (total supply) are minted and sent to the Initial Admin (Safe Multisig) within the contract creation transaction.
        - Verify GOATAI_ERC20 on Basescan.
    - Deploy **Governor** & **TimelockController** via Tally.xyz
    - Deploy OpenZeppelin’s **VestingWallet** for the Community Treasury. The TimelockController will be the recipient of the funds.
        - Verify the VestingWallet contract on Basescan.

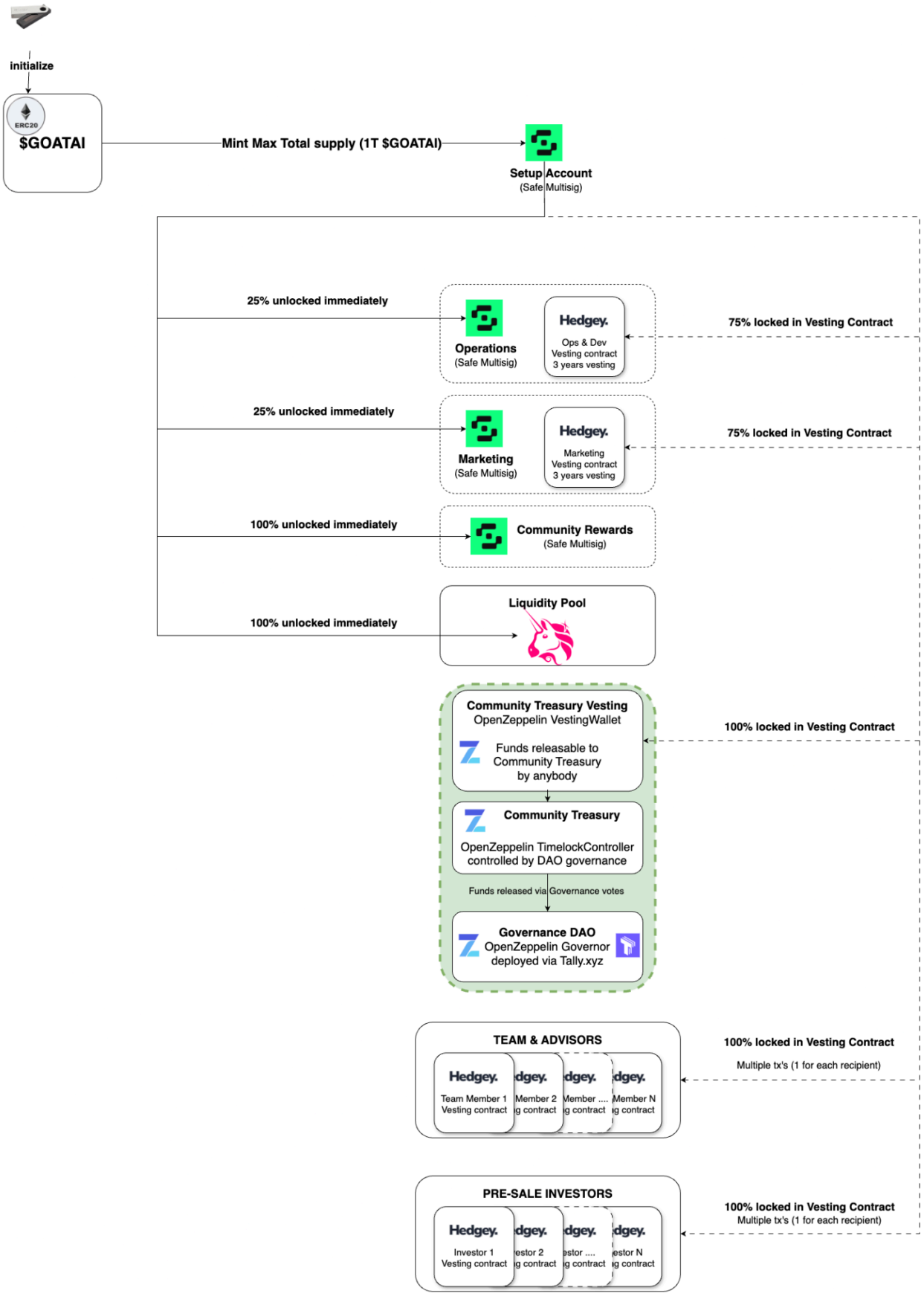
3. Vesting contracts creation
    - On Hedgey Finance, create the vesting contracts for these recipients:
        - Operations
        - Marketing
        - Team & Advisors (1 per member)
        - Pre-sale investors (1 per investor)

    - The Initial Admin sends the locked GOATAI tokens amount to each vesting contract.

4. Unlocked tokens distribution
    - Distribute unlocked tokens according to tokenomics to these entities:
        - Operations
        - Marketing
        - Community Rewards

5. Liquidity Pair creation on Uniswap
    - Create WETH-GOATAI Liquidity Pair on Uniswap. Select V2 contracts since they support fee-on-transfer tokens (and V3 does not).
    - Add a tiny amount of liquidity (e.g. < USD 1) so as to not attract sniping bots before the fee-detection can work.
    - Execute ***setLPPair***, using the newly created Liquidity Pair address. This lets the contract distinguish buy/sell fees from regular transfers.

- Add full liquidity amount (as per tokenomics) to Uniswap.


initialize
ERC20
$GOATAI
Mint Max Total supply (1T $GOATAI)
Setup Account
(Safe Multisig)
25% unlocked immediately
Operations
(Safe Multisig)
Hedgey.
Ops & Dev
Vesting contract
3 years vesting
75% locked in Vesting Contract
25% unlocked immediately
Marketing
(Safe Multisig)
Hedgey.
Marketing
Vesting contract
3 years vesting
75% locked in Vesting Contract
100% unlocked immediately
Community Rewards
(Safe Multisig)
Liquidity Pool
100% unlocked immediately
Community Treasury Vesting
OpenZeppelin VestingWallet
Funds releasable to
Community Treasury
by anybody
100% locked in Vesting Contract
Community Treasury
OpenZeppelin TimelockController
controlled by DAO governance
Funds released via Governance votes
Governance DAO
OpenZeppelin Governor
deployed via Tally.xyz
TEAM & ADVISORS
Hedgey.
Team Member 1
Vesting contract
Member 2
Member ....
Member N
100% locked in Vesting Contract
Multiple tx's (1 for each recipient)
PRE-SALE INVESTORS
Hedgey.
Investor 1
Vesting contract
100% locked in Vesting Contract
Multiple tx's (1 for each recipient)

Audit Goals

What We Want Auditors to Focus On:

- Ensure fee-on-transfer logic is correctly implemented and cannot be bypassed.
- Check the security of the GOATAI_ERC20 contract, where custom functions have been written, especially related to fee management.
- Assess security of architectural setup.
- Assess access control to prevent admin abuse

Additional Notes:

Already audited modules and contract templates are to be considered outside the audit scope, although their integration in the GOATAI_ERC20 contract and in the Governance architecture can be considered within scope. These are the modules and integrations we are referring to:

- All OpenZeppelin contracts and modules used for GOATAI_ERC20
- OpenZeppelin Governor + TimelockController & setup by Tally.xyz
- VestingWallet by OpenZeppelin
- Vesting contracts by Hedgey Finance