Growth Defi V1

Date	December 2020
Lead Auditor	John Mardlin
Co-auditors	Alexander Wade

1 Executive Summary

From November to December 2020, Consensys Diligence engaged with Growth DeFi to assess the security of the Growth DeFi v1 smart contracts: a set of tokens forming the backbone of the Growth Defi platform.

The assessment was conducted by John Mardlin and Alexander Wade, and took place over three calendar weeks: from November 23 to December 11, 2020. A total of 5 person-weeks were allocated over this period.

2 Scope

Our review concerned the files at commit `761f0a7af73a082ac64498061749db4466673542`.

This assessment's primary focus was to review code most pertinent to the function of the various Growth DeFi "gTokens." Specifically, we reviewed the Type 0, Type 1, and Type 2 gToken smart contracts.

The following was **out of scope**:

- Type 3 gTokens and their dependencies
- Non-Solidity files

- The Growth DeFi webapp

Additionally, we **deprioritized** review of the contracts' interactions with various DeFi protocols, including, but not limited to:

- AAVE
- Balancer
- Compound
- DyDx
- Sushiswap
- Uniswap

Rather than review the finer points of interaction with several of these external systems, we spent the majority of our time reviewing the inner workings of the various gTokens, as this represented the core of the Growth DeFi system.

3 Trust Model

In any system, it's important to identify what trust is expected/required between various actors. This is particularly important given the anonymous nature of the developer team.

Users of the system must trust the administrators of the system with the following capabilities:

1. Determining which tokens are held in a given `gToken` and their corresponding percentages.
    1. This portfolio can be modified at any time: before or after a user's deposits and withdrawals.
    2. An administrator could list "fake" type1 and type2 tokens which return false data regarding the balances of the underlying reserve.
2. Each `gToken` of type0, type1 and type2 manages a corresponding liquidity pool in a Balancer AMM. The administrator can initiate a withdrawal of all funds from this pool to an arbitrary address. After a 7-day waiting period, the migration and transfer of funds may be completed. Token holders will need to be vigilant for these events.

3. Setting an arbitrary exchange address, which normally would be used to sell the proceeds of staking and yield farming. This capability could be used to simply drain those funds.
4. Setting the collateralization ratio used for lending assets. This does not present an obvious opportunity for profit, but there is a risk that the `gToken`'s lending position could be liquidated due to mismanagement, or inactivity.

3.1 Actors

Whereas the previous section summarized the *consequences* of malicious or incompetent action by the administrators, this section presents an itemized list of the specific actions available to the administrators and users

Owner (administrator):

- **Portfolio management:**
  - `insertToken()`: Add gTokens to the portfolio management token's list of managed tokens
  - `removeToken()`: Remove gTokens form the portfolio management token's list of managed tokens
  - `transferTokenPercent()`: Change the target percentage for each managed token
  - `setRebalanceMargins()`: Modify the amount by which a token is allowed to deviate from its target percentage
- **Pool management:**
  - `allocatePool()`: Initialize a gToken's Balancer liquidity pool and associate it with the gToken contract
  - `setLiquidityPoolBurningRate()`: Allows the Owner to set the Balancer liquidity pool burning rate. This rate is a percent of the gToken's held BPool tokens that can be burned via `GTokenBase.burnLiquidityPortion`. The rate set may be any value between 0 and 2e16, representing 0-2%.
  - `burnLiquidityPoolPortion()`: Allows the Owner to exit the gToken from its associated Balancer pool for a certain percentage of held BPool tokens, burning any received funds. The exit amount is a percentage of held pool tokens given by the gToken's burning rate.

  - `initiateLiquidityPoolMigration()`: Allows the Owner to initiate a migration of assets held in the gToken's associated Balancer pool to an arbitrary address. The migration may be cancelled by the Owner at any time, and can only be completed after a period of 7 days passes.
  - `cancelLiquidityPoolMigration()`: Allows the Owner to cancel a pending pool migration once it has been initiated.
  - `completeLiquidityPoolMigration()`: Allows the Owner to complete the Balancer pool migration process. The gToken exits its associated pool with the entirety of its BPool tokens. The received GRO and gTokens are then transferred to the migration recipient specified by the Owner at migration initiation.
- **Yield farming management:**
  - `setExchange()`: setting the exchange address used to convert yield farming assets (`COMP` and `DAI`) to the underlying asset
  - `setCollateralizationRatio()`: Adjust percent allocation of the reserve token between two managed tokens
  - `setBurningRate()`: Set the burn rate for each gToken
  - `setMiningGulpRange()`: Set the minimum and maximum amount of the mining token to be converted
  - `setRebalanceMargins()`: Set margins for acceptable deviation from the PMT's percent allocation before triggering a rebalance action

User:

- `deposit()` / `depositUnderlying()`: Can deposit their gTokens into the Growth protocol
- `withdraw()` / `withdrawUnderlying()`: Can withdraw their gTokens from the Growth protocol

4 Action Items

Our assessment determined that significant improvement was needed in the following areas:

4.1 Increase overall quality and quantity of testing

Many individual components of Growth DeFi v1 are covered by unit tests, and a stress-testing system exists to simulate random interaction with the protocol. However, the system lacks comprehensive integration and scenario testing.

The existing unit tests are incomplete, and do not cover many important functions and features of the contracts. Some of the contracts not covered by unit tests include, but are not limited to:

- `GLiquidityPoolManager.sol`
- `GPortfolioReserveManager.sol`
- `GADelegatedReserveManager.sol` (note that `GCDelegatedReserveManager` has tests)
- `Flashloans.sol`
- `SushiswapExchangeAbstraction.sol`
- `UniswapV2ExchangeAbstraction.sol`

However, unit tests alone are not sufficient. Especially for a highly-configurable system like Growth DeFi, comprehensive integration testing is needed to ensure that each individually-tested component works as expected with other components. Integration testing should include the inner workings of each `gToken`, but should also extend to the system's interaction with external protocols (like DEXes, Flashloan providers, and other DeFi protocols).

Finally, the highly-complex nature of Growth DeFi demands a complementary battery of scenario tests. Scenario tests describe complete sequences of events in your system, and are often replete with interactions from multiple simulated actors. As an example, one scenario test for a `gToken` may begin with contract initialization, simulate multiple users depositing, simulate multiple users withdrawing, and end with the contract drained of all assets.

Note that this differs from the existing random stress tests: rather than simulating random actions, scenario tests should simulate specific sequences of actions that are likely to occur during regular use of the system.

Recommendation

Implementing a robust, complete test suite requires significant effort and careful consideration outside of the scope of this review.

In general, **write tests that encapsulate the specification.** Tests should address each of a system's requirements. A system's requirements should be clearly defined within the system design specification. Ensure that the Growth DeFi test suite accurately reflects the most up-to-date specification and includes checks for all of the requirements mentioned therein.

4.2 Improve system documentation and create a complete technical specification

A system's design specification and supporting documentation should be almost as important as the sytem's implementation itself.

- Users rely on high-level documentation to understand the big picture of *how a system works*. Without spending time and effort to create palatable documentation, a user's only resource is the code itself, something the vast majority of users cannot understand.
- Security assessments depend on a complete technical specification to understand the specifics of *how a system works*. When a behavior is not specified (or is specified incorrectly), security assessments must base their knowledge in assumptions, leading to less effective review.
- Maintaining and updating code relies on supporting documentation to know *why the system is implemented in a specific way*. If code maintainers cannot reference documentation, they must rely on memory or assistance to make high-quality changes.

Currently, the only documentation for Growth DeFi is a single `README` file, as well as code comments. While significant effort has been invested into the latter, this system's documentation should be considered incomplete until both high-level and low-level descriptions for its behavior exist outside of the codebase itself.

4.3 Ensure system states, roles, and permissions are sufficiently restrictive

Smart contract code should strive to be *strict*. Strict code behaves predictably, is easier to maintain, and increases a system's ability to handle nonideal conditions.

assessment of Growth DeFi found that many of its states, roles, and permissions are loosely defined:

- Growth DeFi's Owner role assigns complete control over a bulk of important system configuration to a single account. This control includes, but is not limited to:
  - Managing Type 0 `gToken` assets under management, by inserting and removing assets at any time.
  - Rebalancing Type 0 `gToken` asset percent distribution
  - Changing the exchange contract used by Type 1 and Type 2 `gcTokens`
- The specific permissions given to the Owner role suggest that future plans to transition this role to a DAO-governed multisig have not been well thought through. In its current configuration, it would be incredibly difficult to transition the management of the Owner's extensive permissions to a DAO-governed multisig.
  - One example of this is the occasional "burn" feature of each `gToken`. Once per week, each `gToken` burns a portion of the funds it holds in its associated Balancer liquidity pool, decreasing the supply of the assets contained within. This function can only be called by the Owner, and can only be called once every 7 days. This means that every 7 days, one signature per `gToken` must be collected from each signer on the Owner multisig. Because there are 34 planned `gTokens`, each signer must commit to providing at least 34 signatures per week.
- Most contracts contain logic that allows the contract to operate on multiple chains (Mainnet, as well as Kovan, Rinkeby, etc)
- Most contracts can be interacted with by users, even if the contract has not been fully initialized, or is in a semi-configured state. For example:
  - All contracts can be interacted with, even if their Balancer liquidity pool has not been created/finalized yet.
  - `GADelegatedReserveManager.adjustReserve` allows users to perform deposits/withdrawals, even if no valid exchange has been set by the Owner.
- Contracts that use flash loans will attempt to use DyDx. If this operation fails, the contracts will attempt to use AAVE, instead.

Recommendation

- **Document the use of administrator permissions.** For users to know what they can expect from Growth DeFi, the administrator's roles and

responsibilities should be clearly and completely documented and communicated.

- **Monitor the usage of administrator permissions.** To ensure the Owner key's operations are not compromised in some way, monitor transactions and events in Growth DeFi for Owner actions.
- **Specify strict operation requirements for each contract.** Define and implement a strict initialization state. If Owner actions may cause the contracts to deviate from this state (for example, by removing the "exchange" in a `gcToken`), determine and document whether users may still interact with the contracts, and if so, what altered behavior they should expect.

5 Recommendations

5.1 Evaluate risks of frontrunning when swapping COMP and DAI to ETH

On any deposit to or withdrawal from a `gcToken`, assets earned via yield farming incentives will be exchanged via an AMM, such as Sushiswap or Uniswap. This action is predictable, which opens it up to front-running attacks causing the `gcToken` to sell its assets below market value.

This is currently mitigated somewhat by the Owner, who can specify minimum and maximum amounts that can be converted at once. However, optimal values for this mitigation will depend on several additional variables, including network gas prices and pool liquidity.

Recommendation

Investigate this risk further. Additionally, the price at which each `gcToken` performs its swaps should be carefully monitored to minimize or avoid losses due to manipulation.

5.2 Evaluate all tokens prior to inclusion in the system

Each `gToken` is concerned with many underlying 3rd-party tokens, and may be dependent on them conforming to the ERC20 standard. Although most token interactions use OpenZeppelin's `SafeERC20` library, this library only protects against the more common deviations from the ERC20 standard.

Review current and future tokens in the system for non-standard behavior. This is a helpful resource outlining known non-standard behaviors. Also consider using `slither-check-erc`.

Particularly dangerous functionality to look for includes a callback (ie. ERC777) which would enable an attacker to execute potentially arbitrary code during the transaction, fees on transfers, or inflationary/deflationary tokens.

5.3 Add parameters to `deposit` and `withdraw` functions to protect users against front-running attacks

Description

The portfolio and reserve value of a `gToken` may change significantly based on the execution of a single transaction. This could result in a user making a deposit to invest in a portfolio with an unexpected composition.

Recommendation

Allow users to specify a minimum number of shares to receive on deposit, and minimum amount to receive on withdrawal.

5.4 Avoid caching the value of `msg.sender`

This variable declaration occurs in many places: `address _from = msg.sender`. We found that it reduced readability by adding an unnecessary line of code, and another variable to keep track of mentally, and recommend avoiding this practice.

5.5 Avoid 'shallow' wrapper functions where possible

Description

The codebase contains many instances where calls to one library simply forward calls to another library with no additional logic.

For example:

- `Transfers._getBalance(token)` obfuscates a simple call to `ERC20(token).balanceOf(address(this))`. The fact that `address(this)` is the balance being queried is obfuscated by this shallow wrapper.

- `Transfers._pullFunds` and `Transfers._pushFunds` wrap `SafeERC20.safeTransferFrom` and `SafeERC20.safeTransfer`, respectively. They also obfuscate the optimization `if (_amount == 0) return`. This is helpful for gas optimization, but whether or not an external call is being made is crucial information that belongs at the call site.

Another example is the use of the `G` library, as in this call to `G.min`

code/contracts/GCTokenBase.sol:L226

```
_underlyingCost = G.min(_underlyingCost, GC.getLendAmount(reserveToken));
```

which simply calls `Math.min`:

code/contracts/G.sol:L21-L23

```
library G
{
        function min(uint256 _amount1, uint256 _amount2) public pure returns
```

The result is that the reader is subjected to frequent context switching in order to understand the actual implementation.

5.6 Use descriptive names for contracts and libraries

The code base makes use of many different contracts, abstract contracts, interfaces, and libraries for inheritance and code reuse.

In principle, this can be a good practice to avoid repeated use of similar code. However, with no descriptive naming conventions to signal which files would contain meaningful logic, we found this codebase difficult to navigate.

During our review we added descriptive prefixes to many of the files:

- interfaces with `I_`
- abstract contracts with `Abs_`
- libraries with `Lib_`

recommend implementing this or a similar convention.

5.7 Remove multi-chain functionality from all contracts

Many contracts contain logic that enables the contracts to operate on multiple networks, including test networks like Kovan, Rinkeby, etc. For example, in `Flashloans.sol`:

code/contracts/modules/FlashLoans.sol:L38-L49

```
function _getFlashLoanLiquidity(address _token) internal view returns (uint2
{
        uint256 _liquidityAmountDydx = 0;
        if ($.NETWORK == $.Network.Mainnet || $.NETWORK == $.Network.Kovan)
                _liquidityAmountDydx = DydxFlashLoanAbstraction._getFlashLoa
        }
        uint256 _liquidityAmountAave = 0;
        if ($.NETWORK == $.Network.Mainnet || $.NETWORK == $.Network.Ropster
                _liquidityAmountAave = AaveFlashLoanAbstraction._getFlashLoa
        }
        return Math._max(_liquidityAmountDydx, _liquidityAmountAave);
}
```

Multi-chain logic significantly increases code complexity, while adding unused branches of execution to production contracts. This practice significantly impedes readability and increases total bytecode size.

Recommendation

Create a set of production-specific contracts without multi-chain functionality. This will prevent accidental misconfigurations, reduce bytecode size, and make it easier to read, review, and understand the code.

5.8 Prevent contracts from being used before they are entirely initialized

Many contracts allow users to deposit / withdraw assets before the contracts are entirely initialized, or while they are in a semi-configured state.

For example:

- `GCTokenType1` and `GCTokenType2` shouldn't be considered initialized until they have been assigned an exchange via `setExchange`. Currently, these contracts can be used, even if the exchange is a zeroed-out address.

- No `gToken` contracts should be publicly usable until a Balancer liquidity pool has been created and finalized.

Because these contracts allow interaction on semi-configured states, the number of configurations possible when interacting with the system makes it incredibly difficult to determine whether the contracts behave as expected in every scenario, or even what behavior is expected in the first place.

Recommendation

- Define and implement an initialization process for each `gToken` variant
- Ensure contracts do not accept user interaction before this process is complete
- Ensure Owner configuration post-initialization leaves the contract in a fully initialized state

5.9 Add a timelock to `onlyOwner` functions to signal configuration changes to users in advance

Most Owner interactions with the contracts allow the Owner to make large changes to contract configuration with no delay or warning.

Recommendation

- Add a timelock to the following functions in Type 0 tokens:
  - `GTokenBase.setLiquidityPoolBurningRate`
  - `GTokenType0.insertToken`
  - `GTokenType0.removeToken`
  - `GTokenType0.transferTokenPercent`
  - `GTokenType0.setRebalanceMargin`
- Additionally, review the `onlyOwner` functions present in other `gToken` variants, and consider adding timelocks to each.

5.10 Evaluate the risks associated with front-running when adding and removing assets to and from liquidity pools

All `gToken` s (there are currently planned to be 34 of them) have an associated Balancer pool to which the `gToken` provides liquidity

Available tokens are added to the pool via `burnLiquidityPoolPortion()` which is an `onlyOwner` protected function. An attacker may monitor for calls to this function and attempt a “sandwich attack”, causing the pool to be imbalanced, and thus receiving fewer liquidity tokens.

6 Findings

Each issue has an assigned severity:

- Minor issues are subjective in nature. They are typically suggestions around best practices or readability. Code maintainers should use their own judgment as to whether to address such issues.
- Medium issues are objective in nature but are not security vulnerabilities. These should be addressed unless there is a clear reason not to.
- Major issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- Critical issues are directly exploitable security vulnerabilities that need to be fixed.

6.1 Potentially dangerous use of a cached exchange rate from Compound Medium

Description

`GPortfolioReserveManager.adjustReserve` performs reserve adjustment calculations based on Compound’s cached exchange rate values (using `CompoundLendingMarketAbstraction.getExchangeRate()`) then triggers operations on managed tokens based on up-to-date values (using `CompoundLendingMarketAbstraction.fetchExchangeRate()`) . Significant deviation between the cached and up-to-date values may make it difficult to predict the outcome of reserve adjustments.

Recommendation

Use `getExchangeRate()` consistently, or ensure `fetchExchangeRate()` is used first, and `getExchangeRate()` afterward.

6.2 Potential resource exhaustion by external calls performed within an unbounded loop Medium

Description

`DydxFlashLoanAbstraction._requestFlashLoan` performs external calls in a potentially-unbounded loop. Depending on changes made to DyDx's `SoloMargin`, this may render this flash loan provider prohibitively expensive. In the worst case, changes to `SoloMargin` could make it impossible to execute this code due to the block gas limit.

code/contracts/modules/DydxFlashLoanAbstraction.sol:L62-L69

```
uint256 _numMarkets = SoloMargin(_solo).getNumMarkets();
for (uint256 _i = 0; _i < _numMarkets; _i++) {
        address _address = SoloMargin(_solo).getMarketTokenAddress(_i);
        if (_address == _token) {
                _marketId = _i;
                break;
        }
}
```

Appendix 1 - Disclosure

ConsenSys Diligence ("CD") typically receives compensation from one or more clients (the "Clients") for performing the analysis contained in these reports (the "Reports"). The Reports may be distributed through other means, including via ConsenSys publications and other distributions.

The Reports are not an endorsement or indictment of any particular project or team, and the Reports do not guarantee the security of any particular project. This Report does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. No Report provides any warranty or representation to any Third-Party in any respect, including regarding the bugfree nature of code, the business model or proprietors of any such business model, and the legal compliance of any such business. No third party should rely on the Reports in any way, including for the purpose of making any decisions to buy or sell any

token, product, service or other asset. Specifically, for the avoidance of doubt, this Report does not constitute investment advice, is not intended to be relied upon as investment advice, is not an endorsement of this project or team, and it is not a guarantee as to the absolute security of the project. CD owes no duty to any Third-Party by virtue of publishing these Reports.

PURPOSE OF REPORTS The Reports and the analysis described therein are created solely for Clients and published with their consent. The scope of our review is limited to a review of Solidity code and only the Solidity code we note as being within the scope of our review within this report. The Solidity language itself remains under development and is subject to unknown risks and flaws. The review does not extend to the compiler layer, or any other areas beyond Solidity that could present security risks. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty.

CD makes the Reports available to parties other than the Clients (i.e., “third parties”) – on its website. CD hopes that by making these analyses publicly available, it can help the blockchain ecosystem develop technical best practices in this rapidly evolving area of innovation.

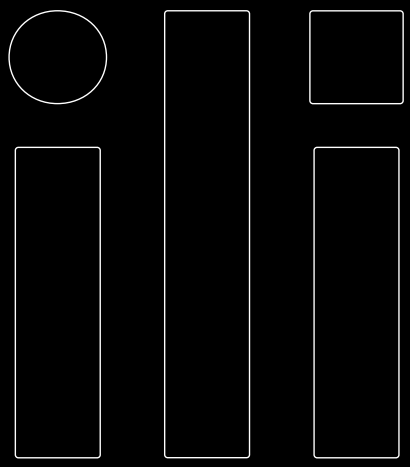
LINKS TO OTHER WEB SITES FROM THIS WEB SITE You may, through hypertext or other computer links, gain access to web sites operated by persons other than ConsenSys and CD. Such hyperlinks are provided for your reference and convenience only, and are the exclusive responsibility of such web sites’ owners. You agree that ConsenSys and CD are not responsible for the content or operation of such Web sites, and that ConsenSys and CD shall have no liability to you or any other person or entity for the use of third party Web sites. Except as described below, a hyperlink from this web Site to another web site does not imply or mean that ConsenSys and CD endorses the content on that Web site or the operator or operations of that site. You are solely responsible for determining the extent to which you may use any content at any other web sites to which you link from the Reports. ConsenSys and CD assumes no responsibility for the use of third party software on the Web Site and shall have no liability whatsoever to any person or entity for the accuracy or completeness of any outcome generated by such software.

TIMELINESS OF CONTENT The content contained in the Reports is current as of the date appearing on the Report and is subject to change without notice. Unless indicated otherwise, by ConsenSys and CD.

Request a Security Review Today

Get in touch with our team to request a quote for a smart contract audit.

CONTACT US



AUDITS
FUZZING
SCRIBBLE
BLOG
TOOLS
RESEARCH
ABOUT
CONTACT
CAREERS
PRIVACY POLICY

Subscribe to Our Newsletter

Stay up-to-date on our latest offerings, tools, and the world of blockchain security.

Email*

e-mail address

→