GFX Interest Protocol

Smart Contract Security Assessment

May. 16, 2022

ABSTRACT

Dedaub was commissioned to perform an audit on GFX's Interest Protocol, at commit hash a90cf72089973ddedb974d23683dc833f896d63d. Two auditors, as well as one trainee, worked on the codebase over two weeks. Our math consultant also contributed to this report.

The GFX team resolved all of the issues and suggestions included in this report, except for the Low Severity issue L5.

Setting & Caveats

The audited codebase is of medium size, at about ~3KLoC.

The audit's main target is security threats, i.e., what the community understanding would likely call "hacking", rather than regular use of the protocol. Functional correctness (i.e. issues in "regular use") is a secondary consideration. Functional correctness relative to low-level calculations (including units, scaling, quantities returned from external protocols) is generally most effectively done through thorough testing rather than human auditing.

Architecture and High-Level Recommendations

Interest Protocol is a lending protocol that establishes a stablecoin (USDi) based on the concept of fractional reserves. USDi is pegged to USD via USDC.
Users of this protocol are able to:
- Provide liquidity: deposit an amount of the reserve token (USDC) and receive an equal amount of USDi
- Borrow USDi: open a borrowing account (Vault) and transfer collateral to it so as to increase its borrowing power in USDi. The protocol allows multi-collateral Vaults for a specific list of assets. Borrowers pay interest on their loan.

- Liquidate Vaults: liquidate an underwater vault to improve its collateral ratio and purchase part of its collateral token(s) in discount paying in USDi
USDi holders share the interest paid by the borrowers except for a small portion of it which is withheld by the protocol as fee.

A complementary part of the protocol is its governance model supported by a separate token called Interest Protocol Token (IPT). IPT holders are able to delegate their voting power. Governance is responsible for deciding and altering the Loan-To-Value (LTV) value of each collateral asset. We believe that this design carries some risks. More specifically, conflicts of interest could arise, for example a group wishing an asset to be highly valued so as to secure their position, or to be poorly valued so that other accounts become liquidatable. Moreover, in cases where the price of a collateral asset falls significantly, the borrowers using this asset are incentivized to vote for increasing its LTV making the state of the protocol even more vulnerable.

The codebase is generally well written and well tested. We were provided with sufficient documentation, namely whitepaper, online docs as well as plenty of explanatory comments within the codebase.

VULNERABILITIES & FUNCTIONAL ISSUES

This section details issues that affect the functionality of the contract. Dedaub generally categorizes issues according to the following severities, but may also take other considerations into account such as impact or difficulty in exploitation:

Category	Description

CRITICAL	Can be profitably exploited by any knowledgeable third party attacker to drain a portion of the system's or users' funds OR the contract does not function as intended and severe loss of funds may result.
HIGH	Third party attackers or faulty functionality may block the system or cause the system or users to lose funds. Important system invariants can be violated.
MEDIUM	Examples: -User or system funds can be lost when third party systems misbehave. -DoS, under specific conditions. -Part of the functionality becomes unusable due to programming errors.
LOW	Examples: -Breaking important system invariants, but without apparent consequences. -Buggy functionality for trusted users where a workaround exists. -Security issues which may manifest when the system evolves.

Issue resolution includes "dismissed", by the client, or "resolved", per the auditors.

CRITICAL SEVERITY:

[No critical severity issues]

HIGH SEVERITY:

[No high severity issues]

MEDIUM SEVERITY:

[No medium severity issues]

LOW SEVERITY:

ID	Description	STATUS
L1	Lack of support for non-standard ERC20 tokens	**RESOLVED**

While the tokens that will be used as collateral are added via governance and are thus expected to be compatible, the protocol does not support tokens deviating from the ERC20 standard by not returning a boolean success value using returndata (USDT is the most popular such token). We suggest the use of the OZ SafeERC20 wrapper library to support both standard and non-standard token implementations.

L2	Potential overflow error in `repayAllUSDi`	**RESOLVED**

In `VaultController::repayAllUSDi`, the following computation takes place to calculate the USDi liability:

```
function repayAllUSDi(uint96 id) external override paysInterest whenNotPaused {
    // grab the vault by id if part of our system. revert if not
    IVault vault = getVault(id);
    // get the total usdi liability, equal to the interest factor * vault's base
liabilty
    // Dedaub: Truncation should happen before the safeu192 call, to prevent any
edge-case overflows.
    uint256 usdi_liability = truncate(safeu192(_interest.factor *
```

```
vault.baseLiability()));
    ...
}
```

However, in this case, truncation takes place after the `safeu192` conversion, which could cause this computation to revert in some edge cases. It's recommended that the operation order is flipped.

L3	Registration status of liquidation asset is not checked	**RESOLVED**

In VaultController::liquidateAccount the asset to be liquidated is not explicitly required to be registered, though this is implied when requesting its current price from oracleMaster:

```
/// VaultController::_liquidationMath
uint256 price = _oracleMaster.getLivePrice(asset_address);

/// OracleMaster::getLivePrice
require(_relays[token_address] != address(0x0), "token not enabled");
```

It is expected that only relays for the registered tokens are deployed, so the above instruction should typically fail in case of an unregistered asset_address. However, there is no guarantee that the available relays are always in sync with Vaultcontroller's registered tokens.

There is an edge case where the above omission could allow an attacker to drain any amount of unregistered token from a vault (probably transferred to the vault by mistake):
- Assume an underwater vault V that has a non-zero balance of token A.
- Assume that A is unregistered but there are relays deployed so that oracleMaster does not fail and returns A's price
- The vault's minter cannot withdraw A because V is underwater
- Now an attacker liquidator can choose to liquidate V as for asset A. The attacker can drain the whole vault's balance of A, since it is not contributing to the vault's borrowing power.

We suggest checking for registered asset_address in liquidateAccount() as a best practice. We also suggest considering allowing withdrawals of unregistered tokens even if the vault is underwater.

L4	Vaults paralyze in case a registered asset returns zero price	**RESOLVED**

In the edge case where a registered asset's price falls to zero, oracleMaster::getLivePrice reverts:

```
function getLivePrice(address token_address) external view override returns
(uint256) {
  require(_paused[token_address] == false, "relay paused");
  require(_relays[token_address] != address(0x0), "token not enabled");
  IOracleRelay relay = IOracleRelay(_relays[token_address]);
  uint256 value = relay.currentValue();
  require(value != 0, "value is 0"); //Dedaub: reverts on 0 price
  return value;
}
```

However, VaultController::get_vault_borrowing_power is not consistent to this design since it would simply omit a token in case of zero price:

```
function get_vault_borrowing_power(IVault vault) private view returns (uint192) {
  uint192 total_liquidity_value = 0;
  for (uint192 i = 1; i <= _tokensRegistered; i++) {
    // ...
    uint256 balance = vault.tokenBalance(token_address);
      if (balance == 0) {
        continue;
      }

    uint192 raw_price = safeu192(_oracleMaster.getLivePrice(token_address));
    // Dedaub: zero prive would revert in oracleMaster
    if (raw_price == 0) {
      continue;
    }
    // ...
```

```
  }
  return total_liquidity_value;
}
```

Therefore, in such an extreme case, vaults that involve a problematic asset A, would practically paralyze since a vault's functionality relies on the calculation of its borrowing power. What is more, minters whose vaults do not hold amounts of A but are close to underwater, have incentives to transfer an amount of A to their vault so as to avoid getting liquidated.

One possible solution would be to remove the problematic asset from the registered assets list, however that whitelist is designed to be append-only (`_tokenAddress_tokenId`). The team informed us that in practice, governance can implicitly remove an asset from the platform by setting its collateral status parameter to 0. `VaultController::get_vault_borrowing_power` should check upon this parameter's value in order to overcome such problematic situations.

L5	Protocol gas consumption increases monotonically, as more collateral tokens are being supported	**DISMISSED**

The core contract of the protocol, `VaultController`, maintains a list of all tokens that can be used as collateral. While this list contains all currently enabled tokens, it's append only, meaning that if governance decides in the future that a token will no longer be supported as collateral (by setting its LTV, liquidation incentive and oracle address to zero), the code will still loop over such tokens, leading to wasted gas.

This issue is prominent in most interactions with a Vault, because a loop over all registered collateral tokens is required (leading to some storage accesses for each) in order to calculate its borrowing power:

```
function get_vault_borrowing_power(IVault vault) private view returns (uint192) {
    uint192 total_liquidity_value = 0;

  //Dedaub: if tokensRegistered grows long, gas cost would increase significantly
  //Dedaub:consider depositing through vault and keep track of each vault's assets
```

```
    for (uint192 i = 1; i <= _tokensRegistered; i++) {
      address token_address = _enabledTokens[i - 1];
      uint256 balance = vault.tokenBalance(token_address);
      if (balance == 0) {
        continue;
      }
      // ...
    }
    return total_liquidity_value;
}
```

Each Vault will most probably hold amounts of only a small subset of the registered tokens. Consequently, the gas costs for interacting with a Vault increases as more tokens are being registered. From a security standpoint, this could even lead to a DoS issue in case the list of registered tokens grows significantly. The fact that registered tokens cannot be completely removed from the list, as per the current design, makes this issue even more concerning.

The following two items are highly recommended and should be considered by the protocol developers, as it would allow for a much more scalable design system (in terms of gas cost):

- Prune the enabledTokens list whenever a token is deprecated
- Maintain a per-account token list, instead of a global one – the average user will only deposit a subset of the supported tokens as collateral.

L6	Anyone can issue arbitrary permits with `address(0x0)` as the signatory/owner.	**RESOLVED**

In `UFragments::permit(owner, ...)` one can issue arbitrary permits if owner is the null address. This is because the precompiled ecrecover returns zero if the message signature is invalid:

```
function permit(owner, ...) {
    // Dedaub: This check is trivial to pass with owner == address(0x0)
    require(owner == ecrecover(digest, v, r, s));
    ...
}
```

While this behavior is really inconsequential to the rest of the protocol (as it is impossible for the null-address to hold any USDi in the current implementation), it is definitely weird and could become an issue in future versions of the protocol.

We highly recommend that an additional require is added to verify that ecrecover terminated successfully, by simply checking that its return value is non-zero.

OTHER/ADVISORY ISSUES:

This section details issues that are not thought to directly affect the functionality of the project, but we recommend considering them.

ID	Description	STATUS
A1	Wrong event emission	**RESOLVED**

In TokenDelegate.sol there is a wrong event emitted upon changing the token's symbol:

```
 function changeSymbol(string calldata symbol_) external override
onlyOwner {
    require(bytes(symbol_).length > 0, "changeSymbol: length invaild");
    // Dedaub: wrong event emitted, should be "ChangedSymbol"
    emit ChangedName(symbol, symbol_);
    symbol = symbol_;
  }
```

A2	Dead code	**RESOLVED**

In GovernorDelegate.sol functions `add256/sub256` are used for protection of overflows/undeflows. However, the solidity compiler of versions ^0.8.0 takes care of these cases automatically. We suggest removing these two functions.

A3	Governance-only modifier	**RESOLVED**

In GovernorDelegate.sol there are a number of functions that are accessible only via the contract itself as a result of governance proposal execution. The access restriction is guaranteed by the following requirement:

```
require(_msgSender() == address(this), "must come from the gov.");
```

We suggest turning this to a modifier to avoid code duplication but also improve readability.

A4	Misleading comments	**PARTIALLY RESOLVED**

1. In `GovernorDelegate.sol`, there is an outdated comment:

```
  /**
   * @notice Used to initialize the contract during delegator contructor
   * @param ipt_ The address of the COMP token // Dedaub: should be IPT token
   * @param votingPeriod_ The initial voting period
   * @param votingDelay_ The initial voting delay
   * @param proposalThreshold_ The initial proposal threshold
   * @param proposalTimelockDelay_ The initial proposal holding period
   */
  function initialize(...) {...}
```

2. In `VaultController.sol`, there are two typos in two comments referring to LTV:

```
function checkAccount(...) {
    ...
```

```
    // if the TLV >= liability, the vault is solvent // Dedaub: should be LTV
    return (total_liquidity_value >= usdi_liability);
}
```

```
function get_vault_borrowing_power(...) {
    ...
        // increase the TLV of the vault by the token value // Dedaub: should be LTV
        total_liquidity_value = total_liquidity_value + token_value;
    ...
}
```

We recommend updating these comments for readability.

A5	CurveMaster is unnecessarily generic	DISMISSED

The current implementation of CurveMaster supports the registration of multiple interest rate curves, as it maintains a mapping from token addresses to curves. However, this generality is unnecessary as the protocol always invokes the CurveMaster functions with the zero address as an argument, which makes sense as only it only needs a single global interest curve.

It is recommended that the code be simplified so that it only keeps track of a single curve, in order to improve readability and reduce gas consumption.

A6	Sanity checks in curve contract constructor	RESOLVED

In the whitepaper, it is mentioned that the interest curve is decreasing. However, the no checks are performed in the curve constructor to ensure that these properties hold:

```
constructor(
    int256 r0,
    int256 r1,
    int256 r2,
    int256 s1,
    int256 s2
) {
```

```
    // Dedaub: No sanity checks on the curve parameters
    _r0 = r0;
    _r1 = r1;
    _r2 = r2;
    _s1 = s1;
    _s2 = s2;
}
```

It is recommended that some basic sanity checks are done to ensure that the parameters of the curve satisfy the property. The checks could include:

- Checking that 0 < r2 < r1 < r0
- Checking that 0 < s1 < s2 < 1e18

A7	Vault: `transferFrom` can be simplified to `transfer`	**RESOLVED**

In `Vault::withdrawErc20`, there a transferFrom operation takes place:

```
  function withdrawErc20(address token_address, uint256 amount) external override
onlyMinter {
      // transfer the token to the owner
      // Dedaub: Can be simply IERC20(...).transfer(...)
      IERC20(token_address).transferFrom(address(this), _msgSender(), amount);
      //  check if the account is solvent
      bool solvency = _master.checkAccount(_vaultInfo.id);
      require(solvency, "over-withdrawal");

      emit Withdraw(token_address, amount);
  }
```

However, since from=address(this), the operation can be replaced by the simpler transfer.

A8	Inconsistent coding style	**RESOLVED**

In VaultController::repayUSDi constant variable ‘expScale’ could be used instead of hardcoding value 1e18:

```
function repayUSDi(uint96 id, uint192 amount) external override paysInterest
whenNotPaused {
    // ...
    // Dedaub: use 'expScale' instead of 1e18 for consistency
    uint192 base_amount = safeu192((amount * 1e18) / _interest.factor);
   // ...
}
```

A9	Unused return data	**RESOLVED**

Function GovernorDelegate::executeTransaction returns the data of a proposal's transaction execution but is never used.

A10	Immutable variables	**PARTIALLY RESOLVED**

In Vault.sol variables `_vaultInfo` and `_master` are set during the contract's construction and cannot be altered, thus they can be declared immutable for readability and gas savings.

A11	Fields are redundantly declared public	**RESOLVED**

In VaultControllers the following fields are redundantly declared public, as getter functions are defined explicitly in the code:

```
// Dedaub: No need to be public - getters defined explicitly in the code
uint256 public _tokensRegistered;
uint192 public _totalBaseLiability;
uint192 public _protocolFee;
```

A12	Unused variables	**RESOLVED**

In UFragments.sol variables `rebasePausedDeprecated` and `tokenPausedDeprecated` are declared and initialized but never used.

A13	Collateral parameters could be incompatible	RESOLVED

In `VaultController`'s `updateRegisteredErc20` and `registerErc20` functions, the parameters of the specified collateral token are written to storage verbatim, including its LTV and liquidation incentive. However, not all (LTV, liquidation incentive) pairs are compatible. More specifically, the following calculation takes place during liquidations:

```
function _liquidationMath(...) {
    uint256 badFillPrice = truncate(price * (1e18 -
_tokenAddress_liquidationIncentive[asset_address]));

    // the ltv discount is the amount of collateral value that one token provides
    uint256 ltvDiscount = truncate(price *
_tokenId_tokenLTV[_tokenAddress_tokenId[asset_address]]);
    // this number is the denominator when calculating the max_tokens_to_liquidate
    // it is simply the badFillPrice - ltvDiscount
    // Dedaub: This implies that the LTV must be < (1 - liquidation incentive) .
    //         Otherwise, the calculation would underflow.
    uint256 denominator = badFillPrice - ltvDiscount;
    // [...]
}
```

It is recommended that a check be added in both `updateRegisteredErc20` and `registerErc20` to verify that LTV < (1 - Liquidation Incentive).

A14	Calculation can be simplified in `VaultController::borrowUsdi`	RESOLVED

In `VaultController::borrowUsdi`, the following calculation takes place:

```
function borrowUsdi(uint96 id, uint192 amount){
    uint192 base_amount = safeu192(uint256(amount * expScale) /
uint256(_interest.factor));
    // Dedaub: Redundant -- this is simply 'amount'
    uint256 usdi_amount =  truncate(uint256(_interest.factor) * base_amount);
    // [...]
}
```

The calculation of `usdi_amount` is redundant and its uses can be replaced with `amount`.

A15	Gas optimization in `VaultController::changeProtocolFee`	**RESOLVED**

In `VaultController::changeProtocolFee`, an event is emitted

```
function changeProtocolFee(uint192 new_protocol_fee) external override onlyOwner {
    require(new_protocol_fee < 1e18, "fee is too large");
    _protocolFee = new_protocol_fee;
    // Dedaub: Emit new_protocol_fee to save some gas
    emit NewProtocolFee(_protocolFee);
}
```

Gas savings can be achieved by emitting `new_protocol_fee`, instead of `_protocolFee` which is a storage field.

A16	Storage layout optimizations in `ChainlinkOracleRelay` and `UniswapV3OracleRelay`	**RESOLVED**

In both `ChainlinkOracleRelay` and `UniswapV3OracleRelay`, all storage fields can be declared as immutable, as they are only initialized in the constructor and never written to again.

Furthermore the following fields are unused and can be removed:

- `ChainlinkOracleRelay::_feedAddress`
- `UniswapV3OracleRelay::_poolAddress`

A17	Floating pragma	**RESOLVED**

The floating pragma ^0.8.0 is used in the contracts, allowing them to be compiled with any version 0.8.x of the Solidity compiler. Although the differences between these versions are small, floating pragmas should be avoided and the pragma should be fixed to the version that will be used for the contracts' deployment.

A18	Compiler known issues	**INFO**

Solidity compiler version v0.8.9 has, at the time of writing, some known bugs. We inspected the code and found that it is not affected by these bugs.

DISCLAIMER

Possible governance attacks within the framework of vote delegation are inherent and thus not in scope for this audit. Certainly, any system with trusted delegates can be subverted if the delegates violate the trust and vote in accordance with interests other than those of their delegators.

The audited contracts have been analyzed using automated techniques and extensive human inspection in accordance with state-of-the-art practices as of the date of this report. The audit makes no statements or warranties on the security of the code. On its own, it cannot be considered a sufficient assessment of the correctness of the contract. While we have conducted an analysis to the best of our ability, it is our recommendation for high-value contracts to commission several independent audits, as well as a public bug bounty program.

ABOUT DEDAUB

Dedaub offers technology and auditing services for smart contract security. The founders, Neville Grech and Yannis Smaragdakis, are top researchers in program analysis. Dedaub's smart contract technology is demonstrated in the contract-library.com service, which decompiles and performs security analyses on the full Ethereum blockchain.