PUBLIC

Code Assessment

of the Gearbox V2 Smart Contracts

October 21, 2022

Produced for

Gearbox

by

CHAINSECURITY

Contents

1 Executive Summary

Dear Gearbox team,

Thank you for trusting us to help Gearbox Protocol with this security audit. Our executive summary provides an overview of subjects covered in our audit of the latest reviewed contracts of Gearbox V2 according to Scope to support you in forming an opinion on their security risks.

Gearbox Protocol implements a general-purpose leverage protocol allowing anyone to borrow funds by providing collateral and paying interest.

This review focuses on a system upgrade from the previous version with several simplifications, allowing users to execute multiple actions as part of core functionality.

The most critical subjects covered in our audit are functional correctness, absence of possibilities to use reentrancy, access control and validation that the system can't actively enter an undercollateralized state. While we found that Gearbox V2 provides high security in these areas internally, it is significantly exposed to errors or misunderstandings in the functionality of integrated third-party systems. Reviewing these external systems for correctness was out of scope for this audit. They keep representing a significant risk for Gearbox V2.

During the review time Gearbox Protocol significantly expanded on the in-code documentation and addressed all issues raised by us, resulting in thoroughly documented and well structured code at the final iteration.

The general subjects covered are code complexity, use of uncommon language features, upgradeability, unit testing, documentation, specification, gas efficiency, trustworthiness and error handling. Security regarding all the aforementioned subjects is high, although documentation needs to be updated to reflect the new system behavior.

In summary, we find that the codebase provides a high level of security.

It is important to note that security audits are time-boxed and cannot uncover all vulnerabilities. They complement but don't replace other vital measures to secure a project.

The following sections will give an overview of the system, our methodology, the issues uncovered and how they have been addressed. We are happy to receive questions and feedback to improve our service.

Sincerely yours,

ChainSecurity

1.1 Overview of the Findings

Below we provide a brief numerical overview of the findings and how they have been addressed.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	3
• Code Corrected	2
• Specification Changed	1
Low-Severity Findings	24
• Code Corrected	20
• Specification Changed	2
• Code Partially Corrected	1
• Acknowledged	1

2 Assessment Overview

In this section, we briefly describe the overall structure and scope of the engagement, including the code commit which is referenced throughout this report.

2.1 Scope

The assessment was performed on the source code files inside the Gearbox V2 repository based on the documentation files. The table below indicates the code versions relevant to this report and when they were received.

V	Date	Commit Hash	Note
1	23 February 2022	5e35d31ca6d8071801c85fba7802e7d085dfe2e0	Initial Version
2	28 March 2022	a90b250e58f2259dc7ee6390bd683aab710cf4cd	Updated code before intermediate report
3	27 June 2022	dfe980f60472e0bbcce006bd4b6afb845747e4ab	After Intermediate report
4	06 July 2022	d1d7e45cf77fd0ccb7f5e7846e155137aa9d2f05	Additional changes after Intermediate report
5	01 October 2022	7b74565870d032bbe7324e7d3f2d1701f4824b0e	Refactoring
6	13 October 2022	c6ca919d46dcd82fa69c89316d9ff969e89bd3f6	V2 Core - Release version
7	19 October 2022	c7290c3ef917f456653e7d5151dc610f338a0805	V2 Integrations - Release version

For the solidity smart contracts, the compiler version `0.8.10` was chosen.

The main review was split in two parts, in the second part an updated commit was received to review. An intermediate report was only delivered after the second part completed. Issues uncovered which have been fixed and did no longer exist in the second commit prior to the intermediate report are omitted.

The following smart contracts are in scope of this review (Paths reflect the latest Version of the Code reviewed):

- adapters/uniswap/*
- adapters/yearn/*
- adapters/curve/* excluding CurveV1_DepositZap.sol
- adapters/convex/*
- adapters/lido/*
- adapters/AbstractAdapter.sol
- credit/*
- libraries/*
- oracles/

Especially the contracts in following subfolders remained largely unchanged compared to the previous audit of GearboxV1:

- core/*
- pool/*
- tokens/*

Open issues and Notes reported in the report of Gearbox V1 are not repeated in this report but may still apply. Please refer to report of the GearboxV1 review.

2.1.1 Excluded from scope

- adapter/lido/WstETHGateway.sol
- adapter/lido/WstETHV1.sol
- integrations/*
- interfaces/*
- core/DataCompressor.sol
- test/*
- tokens/DegenNFT.sol
- support/
- pathfinder/

Deployment, including the smart contracts in subdirectory `factories` are not in scope of this review.

2.2 System Overview

This system overview describes the initially received version (**Version 1**) of the contracts as defined in the Assessment Overview.

Gearbox implements a general-purpose leverage protocol for ERC-20 tokens. The system is modular and consists of different parts. This report covers the new Gearbox V2 system. V2 consists of the same modules as V1. All modules work together within a release, some of the modules can be used across releases. The following release-overarching combinations have been considered in this audit:

V2 credit system connecting to a V1 Pool Service and the V1 AccountFactory / Credit Accounts.

Other combinations have not been reviewed. Notably PriceOracle of V1 cannot be used in V2. Adapters count as part of the credit system and must not be used across versions.

2.2.1 The Credit System

It consists of four contracts: The three core contracts CreditFacade, CreditConfigurator and the CreditAccounts.

1. `CreditFacade`: The user interface for the user to interact with the CreditManager. It exposes the following functionalities for users:

    - `openCreditAccount`: Takes a credit account from the stock of accounts and moves all the user's funds and the leverage to the account. Compared to V1, in V2 no `maxLeverageFactor` is enforced, the only restriction is that the health factor must be > 1.

- `openCreditAccountMulticall`: Opens a new credit account, executes the given multicalls before doing a full collateralization check.
- `closeCreditAccount`: Allows a user to close his credit account. This requires repayment of the full debt plus fees of the system. Sends all remaining assets of the credit account to the specified address (The V1 function repayCreditAccount no longer exists). Multicalls may be executed before the actual closure to trade tokens of the CreditAccount into one asset. Tokens can be skipped, WETH can be converted into ETH.
- `liquidateCreditAccount`: It allows any user to liquidate an undercollateralized credit account. The liquidation process first determines the current value of the credit account. The liquidator buys the funds of the credit account under liquidation for a discounted price (percentage based liquidation discount). Using this amount of funds in the underlying (either present at the credit account or must be supplied by the liquidator) the credit account is closed. Multicalls may be executed by the liquidator to swap tokens of the credit account before the closure. During closure the system handles profit (based on the fee on the interest and the liquidation penalty) or loss if the value minus the liquidation discount is unable to cover the debt. Surplus (value exceeding debt plus all fees) is returned to the owner of the credit account in the form of the underlying.
- `increaseDebt`: It further increases the loan taken by the user. The new debt may not exceed `maxBorrowedAmount`.
- `decreaseDebt`: It decreases the loan taken by the user, funds for this operation must be available at the credit account. The new debt amount of this credit account may not be below `minBorrowedAmount`.
- `addCollateral`: Adds an amount of a supported token to the credit account, enables the token for this credit account
- `multicall`: Allows to bundle actions. Executes the internal `_multicall` function which is also used by the respective close/liquidate/open`` functions. Multicalls works as follows: A multicall consists of a target address and callData. Allowed target address are the adapter contracts supported by the CreditManager or the CreditFacade. Only certain functions for basic operations on the credit account are supported for calls to the credit facade (increase/decrease debt, add collateral.) Debt may not be decreased when it has been increased already within this multicall. For multicalls executed prior to closure/liquidations multicalls to the CreditFacade are blocked, only multicalls to the adapters are allowed.

Furthermore, functions managing the transfer of credit account ownership and a function allowing the credit accounts owner to give token transfer allowance to a supported target contract is present.

2. `CreditConfigurator`: Used by the configurator role to configure the credit manager.
3. `CreditManager`: Core contract. Each CreditManager defines an underlying token and is connected to a liquidity pool with the same underlying token. Users execute it's functionality through the CreditFacade, the CreditConfigurator is used to configure the CreditManager.
4. `CreditAccount`: It represents a leverage position and holds all the position's balances acting essentially as a wallet. The owner's access to this wallet is restricted as it contains additional funds borrowed from the pool. Interaction with external protocols using the funds of the credit account can be executed through the respective adapters.

2.2.2 The Pools

The pools are used to manage the liquidity of the system. Users can lend funds to the pool and accrue interest. The funds held by the pool are then used as leverage by the users that hold credit accounts.

A pool also defines a denomination asset which is used to evaluate the pool's holdings. It exposes to the users the following functionalities:

1. `addLiquidity`: Users transfer an amount of the denomination asset to the pool and mint an amount of Diesel tokens.
2. `removeLiquidity`: The users exchange the diesel token they hold for the corresponding amount of the denomination asset. Note that redemption is not guaranteed at all times as funds may be borrowed to credit accounts.

2.2.3 The Credit Account Factory

In order to reduce costs of the deployment of the credit accounts, an account renting system is implemented. Upon opening a credit account, a free credit account contract is taken from the factory. After a position's closure, the credit account is returned to the factory. In case all the credit accounts are used, a new one is created by using the cloning paradigm.

2.2.4 Checking Collateral

The system calculates the collateralization of a position using the health factor. The health factor is essentially the ratio between a discounted value of the holding of an account and the amount that has been borrowed by the account. The discount in the value aims to prevent abrupt fluctuations in the values of the assets. As long as the health factor is greater than 1, the account is considered healthy. Otherwise, it can be liquidated.

Anyone may liquidate unhealthy credit accounts.

In order to prevent adversarial actions by the users such as stealing part of the collateral, a check is done after each action on the funds at the credit account, e.g., a trade with an external platform. This check on the collateral prevents an action from leaving a credit account undercollateralized.

However, checking the health factor is gas-heavy. In order to avoid this check after each action, Gearbox introduces fast check protection. Fast check protection is another check which limits the decrease in the collateral value. More specifically, it does not allow an action to reduce the collateral value measured in the difference of the spend and incoming assets to reduce more than a specified percentage. An additional safeguard is that after a certain number of fastchecks, a full health check has to be performed.

Note that fast check cannot cover for the edge case when the collateral is close to 1 and a non-profitable trade reduces the health factor under 1.

2.2.5 Adapters

The credit accounts can interact with external protocols via the adapters. The adapters are the only entry points which allow the aforementioned interaction. The adapters currently implemented by the system are the following:

- `UniswapV2` and `UniswapV3` which allow the credit account to trade its holdings for other assets.
- `YearnV2` which allows the credit account to deposit and withdraw assets from yearn vaults
- `CurveV1` which facilitates arbitrage with leverage on tokens which are part pools that the underlying token of the credit account also participates.
- `LidoV1` interaction with Lido.fi

- `Convex` interaction with Convex Finance

Generally, the adapters are implemented to mimic the function interface of the DeFi contract by implementing the functions with the same name and parameters. The adapters process the call on the function of the 3rd party DeFi system this adapter connects to before executing a check on the new state of the credit account. This ensures that the action did not make the credit account unhealthy. The check is only done if the user called the Adapter directly, whenever the call originates from the CreditFacade no check on the collateral is done by the Adapter. The CreditFacade is trusted to perform these checks.

The current assumption of the adapters is that the balance of the asset sent to the external protocol will not increase and the balance of the asset received from the external protocol will not decrease.

Trust Model & Roles

The system relies heavily on the `Configurator` role since they set the parameters of the system with few restrictions. Hence, the configurator is a role trusted by the system and is supposed to act honestly and correctly.

In general, more roles are implemented through the `ACL` which is the common authorization layer shared by the whole system. There, more roles are defined, i.e., the `pausableAdmin` and the `unpausableAdmin` who can pause and unpause the system respectively.

Moreover, the system heavily relies on the prices the Chainlink oracles provide to the system. Should the oracles behave improperly, the system can evaluate the credit accounts erroneously and allow liquidations that should not take place.

Tokens enabled for use in the system are assumed to be non-malicious ERC-20 tokens without special behaviors including but not limited to having no callbacks on transfer and no balance changing actions like rebalancing.

Finally, the system interacts with third-party protocols, namely, `UniswapV2`, `UniswapV3`, `CurveV1`, `YearnV2`, `Lido` and `Convex`. These protocols are assumed to work correctly. Moreover, any malfunction of these protocols can seriously compromise the security and the correct behavior of the system.

Curve pool: Fully trusted to work correctly. The Curve price feeds only works for Curve Pools version 2. In particular version 1 pools and tricrypto-style pools are not supported by this price feed. Furthermore, the use of the virtual price implies the assumption that the pools are balanced, which they will not be at all points in time.

In case Curve-Lending Pools are used, the protocol receiving the actual funds are also trusted to work correctly. Incorrectly set-up pools, e.g. certain trustless factory-produced pools, will not work correctly.

Users are generally untrusted.

2.2.6 Changes in Version 3

Version 3 of the code includes the following changes which are not directly related to fixes of raised issues:

- Safe launch with DegenMode: A Credit Facade with Degen Mode enabled only allows owner of special account-bound NFTs to open accounts.
- FastCheck takes liquidation thresholds into account: It now ensures that the HF has not decreased significantly rather than the pure collateral value. The decrease is tracked cumulatively across multiple swap and as soon as liqudiationFee of cumulative loss is occurred, a full collateral check is performed and the cumulative sum is reset.

2.2.7 Changes in Version 4

- Multicall now supports `revertIfBalanceLessThan` allowing users to add additional slippage protection.

2.2.8 Changes in Version 5

- Universal adapter was introduced. This adapter allows users to arbitrarily revoke approvals to given contracts. The universal adapter differs from the other adapters since it does require to specify a target contract.
- The code was extensively refactored.
- The maximum allowed number of enabled tokens per credit account is now limited to 12.

2.2.9 Changes in Version 6 & 7

- Final refactoring and split into core, integrations and contracts repositories.
- Fixed issues highlighted in Version 5

3 Monitoring

A thorough code audit is just one important part of a comprehensive smart contract security framework.

Next to proper documentation/specification, extensive testing and auditing pre-deployment, security monitoring of live contracts can add an additional layer of security. Contracts can be monitored for suspicious behaviors or system states and trigger alerts to warn about potential ongoing or upcoming exploits.

Consider setting up monitoring of contracts post-deployment. Some examples (non-exhaustive) of common risks worth monitoring are:

1. Assumptions made during protocol design and development.
2. Protocol-specific invariants not addressed/mitigated at the code level.
3. The state of critical variables
4. Known risks that have been identified but are considered acceptable.
5. External contracts, including assets your system supports or relies on, that may change without your knowledge.
6. Downstream and upstream risks - third-party contracts you have direct exposure to (e.g. a third party liquidity pool that gets exploited).
7. Privileged functionality that may be able to change a protocol in a significant way (e.g. upgrade the protocol). This also applies to on-chain governance.
8. Protocols relying on oracles may be exposed to risks associated with oracle manipulation or staleness.

We have identified some areas in Gearbox V2 that would be well suited for security monitoring. As some monitoring rules are only powerful if an attacker isn't aware of them, we ommit specific rules which we provide in our confidential reports from our public report.

4 Limitations and use of report

Security assessments cannot uncover all existing vulnerabilities; even an assessment in which no vulnerabilities are found is not a guarantee of a secure system. However, code assessments enable the discovery of vulnerabilities that were overlooked during development and areas where additional security measures are necessary. In most cases, applications are either fully protected against a certain type of attack, or they are completely unprotected against it. Some of the issues may affect the entire application, while some lack protection only in certain areas. This is why we carry out a source code assessment aimed at determining all locations that need to be fixed. Within the customer-determined time frame, ChainSecurity has performed an assessment in order to discover as many vulnerabilities as possible.

The focus of our assessment was limited to the code parts defined in the engagement letter. We assessed whether the project follows the provided specifications. These assessments are based on the provided threat model and trust assumptions. We draw attention to the fact that due to inherent limitations in any software development process and software product, an inherent risk exists that even major failures or malfunctions can remain undetected. Further uncertainties exist in any software product or application used during the development, which itself cannot be free from any error or failures. These preconditions can have an impact on the system's code and/or functions and/or operation. We did not assess the underlying third-party infrastructure which adds further inherent risks as we rely on the correct execution of the included third-party technology stack itself. Report readers should also take into account that over the life cycle of any software, changes to the product itself or to the environment in which it is operated can have an impact leading to operational behaviors other than those initially determined in the business specification.

5 Terminology

For the purpose of this assessment, we adopt the following terminology. To classify the severity of our findings, we determine the likelihood and impact (according to the CVSS risk rating methodology).

- *Likelihood* represents the likelihood of a finding to be triggered or exploited in practice
- *Impact* specifies the technical and business-related consequences of a finding
- *Severity* is derived based on the likelihood and the impact

We categorize the findings into four distinct categories, depending on their severity. These severities are derived from the likelihood and the impact using the following table, following a standard risk assessment procedure.

Likelihood	**Impact**		
	High	Medium	Low
High	**Critical**	**High**	**Medium**
Medium	**High**	**Medium**	**Low**
Low	**Medium**	**Low**	**Low**

As seen in the table above, findings that have both a high likelihood and a high impact are classified as critical. Intuitively, such findings are likely to be triggered and cause significant disruption. Overall, the severity correlates with the associated risk. However, every finding's risk should always be closely checked, regardless of severity.

6 Findings

In this section, we describe any open findings. Findings that have been resolved have been moved to the Resolved Findings section. The findings are split into these different categories:

- Security: Related to vulnerabilities that could be exploited by malicious actors
- Design: Architectural shortcomings and design inefficiencies
- Correctness: Mismatches between specification and implementation

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	0
Low-Severity Findings	2

- Curve Price Oracle Manipulation Acknowledged
- Enable Supported Token on Any CreditAccount Code Partially Corrected

6.1 Curve Price Oracle Manipulation

Security Low Version 1 Acknowledged

When estimating the value of a Curve LP token, `get_virtual_price()` is queried which describes the value increase through fees since the pool was created. The function may return a manipulated value for some pools where transfers have callbacks or other callbacks to users are made (e.g. ETH, ERC677, ERC223, ERC777, ...) as the state of the pool may be inconsistent during the callback.

Due to the limiter of the Pricefeed which enforces that the price remains within a certain bound, Gearbox is largely protected hence the low severity rating. Nevertheless, the manipulated state of the Curve pool could be detected (at this point the pool's reentrancy lock is set) by the pricefeed.

Curve is aware of this issue and new pools are no longer affected. Existing pools however remain vulnerable. This issue is currently being addressed and affects various integrations. As of today not all have been fixed hence please treat this issue confidential for the time being. Full public disclosure is expected to be released soon.

Gearbox Protocol responded as follows:

> Due to the LP price limiters, the attacker cannot practically inflate the asset value more than 2% of its real value. This discrepancy can be included in the asset's LT - the LT represents the maximal asset price drop during the liquidation period, but is not dependent on whether this price drop comes from actual market conditions, or price manipulation within a bound known in advance.

6.2 Enable Supported Token on Any CreditAccount

Security **Low** **Version 1** **Code Partially Corrected**

`CreditFacade.addCollateral()` allows anyone to deposit funds on behalf of another credit account owner. Using this function has a different effect compared to simply transferring the funds to the credit account directly: It additionally enables the token for this credit account.

This may be a risk factor: If there is ever a bad token supported by a CreditManager, this immediately affects all CreditAccounts of this CreditManager. Users are not safe when they don't hold the affected token.

Code partially corrected:

`CreditFacade.addCollateral()` is now only allowed for users for which are authorized in the `transferAllowed` mapping.

Gearbox Protocol notes:

> This change should address an attacker sending a bad token to other users in order to break health factor calculation. However, there still remains a narrow vector whereas a token that was already on Credit Account is broken and reverts on balanceOf() (for example, stETH and SNX use proxies, and can be potentially changed to a broken implementation contract).
>
> Currently, this is not addressed, however, should this transpire, CreditFacade could be quickly updated to ignore this token (or error-handle) in calcTotalValue(), which would allow to liquidate affected positions.

7 Resolved Findings

Here, we list findings that have been resolved during the course of the engagement. Their categories are explained in the Findings section.

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0

High-Severity Findings	0

Medium-Severity Findings	3

- Multicall Actions During Pauses Code Corrected
- Pricefeed of Oracle May Be Updated Specification Changed
- Unable to Handle Missing Return Value Code Corrected

Low-Severity Findings	22

- CumulativeDrop Calculation Rounding Code Corrected
- twvUSD Contains Value in Underlying Code Corrected
- Free Flashloan upon Open/Close Code Corrected
- Adapters Ignore User Input Code Corrected
- Add Token Without Liquidation Threshold Code Corrected
- Checking for Valid Token Indices for Curve Pools Is Too Loose Code Corrected
- Credit Accounts Give Very High Approval to Contract Code Corrected
- CreditAccount Calls approve() on Unsupported Token Code Corrected
- Curve Registry Code Corrected
- CurveV1 Adapters: TokenOut Might Not Be Enabled at CreditAccount Code Corrected
- Duplicate Error Code Used Code Corrected
- Incorrect Comment After Refactoring Code Corrected
- Incorrect Descriptions Specification Changed
- Outdated Function Description Specification Changed
- PriceOracle: Unused Timestamp Code Corrected
- Read-only Reentrancy Code Corrected
- Redundant Event Emission Code Corrected
- Redundant Initialization Code Corrected
- Reentrancy Into CreditFacade Acknowledged Code Corrected
- Sanity Check of New Pricefeed Code Corrected
- Unused allowedContractsSet Code Corrected
- YearnV2Adapter: Different Behavior of Functions Code Corrected

7.1 Multicall Actions During Pauses

Design Medium Version 1 Code Corrected

During a liquidation, the liquidator can call `CreditFacade._multicall`. When this happens, the ownership of the credit account is temporarily passed to the `CreditFacade` to allow it to properly interact with the adapters. By using this feature, liquidators can swap tokens of the credit account to the underlying and, thus, they don't have to supply the underlying by themselves. This is a useful feature for any liquidator, even for the emergency ones. During pauses, however, the functionality of the credit manager is limited. One of the limitations is that `CreditManager.transferAccountOwnership` fails. This means that the liquidators cannot make any calls to the adapters.

Code corrected:

The `CreditManager` now allows multiple calls to be made while the system is paused as long as the call is related to an emergency liquidation (`whenNotPausedOrEmergency`).

7.2 Pricefeed of Oracle May Be Updated

Correctness Medium Version 1 Specification Changed

`PriceOracle._addPriceFeed()` is annotated with

```
/// @dev Sets price feed if it doesn't exist. If price feed is already set, it changes nothing
/// This logic is done to protect Gearbox from priceOracle attack
/// when potential attacker can get access to price oracle, change them to fraud ones
/// and then liquidate all funds
```

The function does not enforce this, a second call to this function allows to update the pricefeed for the token.

Specification changed:

The description was erroneous and it was intended for the function to update the existing price feed. The description was updated to reflect that.

7.3 Unable to Handle Missing Return Value

Design Medium Version 1 Code Corrected

Compared to Gearbox V1 `safeApprove()` has been replaced by `approve()` in function `CreditAccount.approveToken`. The interface inherited expects a boolean return value as defined in the ERC-20 specification. However, there are tokens such as USDT or OMG which do not adhere to this specification and have no return value on `approve()` and `transfer`.

Calling `CreditAccount.approveToken()` with these tokens will revert as the function call does not return the expected return value. Hence it's not possible for the new credit accounts to give approval on such tokens.

Code corrected:

The new CreditAccount implementation no longer features an `approveToken` function. Approvals through `CreditManager.approveCreditAccount()` now use the `execute` function of the CreditAccount which allows arbitrary calls. This works for both, the new implementation and the old already deployed credit accounts.

If present, the returned boolean is checked. In case the approval is unsuccessful the code attempts to reset the approval to 0 before attempting the to approve the intended amount. This accounts for some token implementations enforcing this.

7.4 CumulativeDrop Calculation Rounding

Correctness Low Version 3 Code Corrected

The new fast collateral check is described as follows:

```
The fast check now ensures that the HF has not decreased significantly, rather
than pure collateral value. The decrease is also tracked cumulatively across
multiple swaps (hence the sum) - as soon as liquidationFee of cumulative loss is
occurred, a full collateral check is performed and the cumulative sum is reset.
```

The computation is done as follows:

```
// compute cumulative price drop in PERCENTAGE FORMAT
uint256 cumulativeDrop = PERCENTAGE_FACTOR -
    ((amountOutCollateral * PERCENTAGE_FACTOR) / amountInCollateral) +
    cumulativeDropAtFastCheck[creditAccount]; // F:[CM-36]

...

if (cumulativeDrop <= slot0.feeLiquidation) {
    cumulativeDropAtFastCheck[creditAccount] = cumulativeDrop; // F:[CM-36]
    return;
}
```

PERCENTAGE_FACTOR is 10`000. This allows precision up to 2 decimals. Resulting rounding errors per division might be up to 0.009999% Drops up to 0.009999...% per fast check are not detected nor added to `cumulativeDropAtFastCheck`. This may be done repeatedly.

Hence the requirement

```
The decrease is also tracked cumulatively across multiple swaps (hence the sum) - as
soon as liquidationFee of cumulative loss is occured, a full collateral check is performed
```

strictly speaking does not hold. Other protocols, e.g. Maker work with significant higher precision internally.

Is the resulting precision sufficient / can the potential loss be tolerated?

The new fast check compares `cumulativeDrop` and `feeLiquidation`. While both are percentages, they are different: The feeLiquidation will be taken from the actual total value while the cumulative drop has been calculated taking into account the liquidation thresholds. Given the liquidation thresholds are strictly lower than 100% there is a safety margin before the system takes a loss.

Code corrected:

The precision of the calculation was increased in RAY. The relevant code snippet now looks like this:

```
// compute cumulative price drop in WAD FORMAT
        uint256 cumulativeDropRAY = RAY -
            ((amountOutCollateral * RAY) / amountInCollateral) +
            cumulativeDropAtFastCheckRAY[creditAccount]; // F:[CM-36]

        // if it drops less that feeLiquiodation - we just save it till next check
        // otherwise new fullCollateral check is required
        if (
            cumulativeDropRAY <=
            (slot0.feeLiquidation * RAY) / PERCENTAGE_FACTOR
        ) {
            cumulativeDropAtFastCheckRAY[creditAccount] = cumulativeDropRAY; // F:[CM-36]
            return;
        }
```

7.5 twvUSD Contains Value in Underlying

Correctness **Low** **Version 3** **Code Corrected**

The public function `CreditFacade.calcCreditAccountHealthFactor()` calculates the health factor in percent:

```
function calcCreditAccountHealthFactor(address creditAccount)
    public
    view
    override
    returns (uint256 hf)
{
    (, uint256 twvUSD) = calcTotalValue(creditAccount); // F:[FA-42]
    (, uint256 borrowAmountWithInterest) = creditManager
    .calcCreditAccountAccruedInterest(creditAccount); // F:[FA-42]
    hf = (twvUSD * PERCENTAGE_FACTOR) / borrowAmountWithInterest; // F:[FA-42]
}
```

The naming of the variable `twvUSD` is misleading since the total weighted value returned by `calcTotalValue()` is in the underlying. Note that it has to be in the underlying for the calculation `hf = (twvUSD * PERCENTAGE_FACTOR) / borrowAmountWithInterest` to be correct.

Code corrected:

`twvUSD` was renamed into `twv`.

7.6 Free Flashloan upon Open/Close

Design **Low** **Version 2** **Code Corrected**

Version 2 introduced a protection which prevents free flashloans by increasing/decreasing debt within the same multicall. A variable within `_multicall()` tracks whether debt has already been increased in this call and if `true` prevents reducing debt.

This prevention however is not effective in a corner case:

- When a new credit account has just been opened through `openCreditAccountMulticall()` debt can be reduced within the multicall (free flashloan).

Code corrected:

The internal variable tracking whether debt has already been increased is now an additional input parameter for `_multicall()`. This allows the calling function `openCreditAccountMulticall()` to pass the information that debt has already been increased and hence the prevention also works in this corner case described in the issue above.

7.7 Adapters Ignore User Input

Design | Low | Version 1 | Code Corrected

The design of the Adapters is that they implement the same interface as the contract they connect to.

Illustrated with the following examples taken from the YearnV2 adapter this issue highlights that user inputs are sometimes silently ignored:

```
/// @dev Deposit credit account tokens to Yearn
/// @param amount in tokens
function deposit(uint256 amount, address)
    external
    override
    nonReentrant
    returns (uint256)
{
    address creditAccount = creditManager.getCreditAccountOrRevert(
        msg.sender
    ); // F:[AYV2-4]

    return _deposit(creditAccount, amount); // F:[AYV2-7,12]
}
```

`deposit()` allows the user to specify the address of the recipient of the yVault shares. Obviously, this is not allowed as the funds must remain with the CreditAccount. The implementation uses "safe defaults", and ignores the user input. This behavior should be documented.

A more critical example is function `withdraw` and parameter `maxLoss`. The user may intend to set the acceptable maxLoss to a lower value. The implementation of the adapter however ignores this value and proceeds with the default. The result may be unexpected for the user.

```
function withdraw(
    uint256 maxShares,
    address,
    uint256 maxLoss
) public override nonReentrant returns (uint256 shares) {
    address creditAccount = creditManager.getCreditAccountOrRevert(
        msg.sender
    ); // F:[AYV2-4]

    return _withdraw(creditAccount, maxShares); // F:[AYV2-9,14]
}
```

Code corrected:

There is now a `withdraw()` override that correctly passes `maxLoss` to the corresponding `withdraw(uint256,address,uint256)` function in the Yearn vault, using an internal _withdrawMaxLoss function.

Note: There are other adapter functions where the inputs are ignored - this happens in 2 cases:

- The adapter passes unmodified msg.data to the target contract, and doesn't need some of the inputs for adapter-specific operations;
- The input is the recipient address, which is always replaced by the credit account address (same cases as the `deposit(uint256,address)` function described in the original issue).

7.8 Add Token Without Liquidation Threshold

Design | Low | Version 1 | Code Corrected

Configurators may add a token to a credit manager using the function `addTokenAllowedList`. Initially, this token has a liquidation threshold of zero, the configurator must set a liquidation threshold using the function `setLiquidationThreshold`.

When no liquidation threshold is set for a token, the balance of this token that the credit accounts hold doesn't count towards the weighted value.

Code corrected:

The function in question is now called `CreditConfigurator.addCollateralToken()`. It now accepts `uint16 liquidationThreshold` as input and calls `_setLiquidationThreshold()` immediately after adding the token. `_setLiquidationThreshold()` checks that the passed LT value is larger than 0.

7.9 Checking for Valid Token Indices for Curve Pools Is Too Loose

Design | Low | Version 1 | Code Corrected

When translating the token index required in Curve pools to the token as known to Gearbox, the following `require` is executed:

```
function _get_token(int128 i) internal view returns (address) {
    require(i <= int128(uint128(N_COINS)), "Incorrect index");
    return (i == 0) ? token0 : token1;
}
```

This check passes for invalid values like negative indices and exactly one index too high, e.g. `i = 2` passes for pools with `N_COINS = 2` like in this example, although only `i = 0` and `i = 1` should pass. While in our understanding Curve will fail when called with invalid tokens, it is safer to ensure that no wrong token indices can be passed to not have to rely on Curve preventing execution with those indices.

Code corrected:

The code has been refactored, the `__getToken()` function in CurveV1AdapterBase is strict and reverts for invalid indices.

7.10 Credit Accounts Give Very High Approval to Contract

Security Low Version 1 Code Corrected

Credit accounts give very high (25% of *uint96.max*) approval to the contracts adapters connect to. This approval remains even when the credit account is returned to the factory or being assigned to the next user. Giving excess approvals introduces a risk: The third party system must be fully trusted and reviewed that these approvals cannot be accessed when not called by the holder of the funds. If this does not hold, e.g. due to a bug, the funds of credit accounts are at risk.

One example where such a bug led to loss of funds: https://medium.com/gelato-network/sorbet-finance-vulnerability-post-mortem-6f8fba78f109

Gearbox uses a trust-minimized approach by validating effects of adapters for token transfers instead of relying on correct execution. The recently discovered UniswapV3 bug would also have been prevented in case targeted allowances are given. Infinite approvals can undermine this approach. Approving only the necessary funds each time also saves gas as the increased allowance is reset to the original value, resulting in a refund which is significantly larger than the overhead cost of calling into a "hot" contract.

Code corrected:

Gearbox Protocol responded:

> For more fine-grained security configuration, there are now 2 allowance models:
>
> 1. Max allowance For highly-trusted protocols (such as Curve or Uniswap) approvals are always set to `type(uint256).max`. For swap-like operations this logic is encapsulated in the `AbstractAdapter._executeMaxAllowanceFastCheck()` function. After each operation, the system returns allowance to the maximal possible value. This significantly improves UX for WalletConnect usage, since users wouldn't have
>
>   to approve tokens in the Uniswap/Curve interface after each transaction.
>
> 2. Limited allowance For other protocols, approvals are set to the available balance on the Credit Account before the operation, and then reset to 1 in the end. This would prevent an attacker from withdrawing assets from Credit Accounts, if they manage to compromise the target contracts.
>
> Maximal and safe allowances for fastCheck operations are set in `AbstractAdapter._executeMaxAllowanceFastCheck()` and `AbstractAdapter._safeExecuteFastCheck()`, respectively. Allowances for fullCollateralCheck operations have to be done manually within adapter functions.

This mitigation still allowed to be circumvented in the following way:

> The limited allowance approach for semi-trusted third-party contracts might be circumvented: Using `CreditFacade.approve()` the current owner of a credit contract may approve a supported token for any supported target contract. Such an approval remains when the credit account is returned to the factory and still exists when the credit account is assigned to the next user.

Gearbox Protocol further improved the security in the following way:

> In order to improve the security of the *CreditFacade.approve()* function, *upgradeableContracts* was added to the Credit Facade. This is a list of contracts with practices potentially detrimental to

security. This includes upgradeable contracts, contracts that can make arbitrary calls (even with admin-only access), etc.

approve now reverts when called on a contract in *upgradeableContracts*. Currently, the Gearbox team intends to include only Lido into the list, as no other supported contracts appear to be upgradeable, or able to call *transferFrom* on CA assets.

To additionally secure assets accounts that don't belong to the attacker but have allowances (e.g., some non-zero allowances may remain after previous use), the first iteration of the Universal Adapter was implemented, which allows users to revoke all allowances on a newly-opened account.

Note: *CreditFacade.approve* is mainly used to support WalletConnect with dApp frontends. Most frontends require non-zero allowance of a token to the contract, and do not allow any further action before *approve* is called. Thus, a function to set allowance separately from adapter actions is required.

7.11 CreditAccount Calls `approve()` on Unsupported Token

Security **Low** **Version 1** **Code Corrected**

`CreditManager.approveCreditAccount()` approves token transfers on behalf of a credit account. The function calls the CreditAccount which then executes a call to the given tokens `approve()` function.

The function is annotated with:

```
/// @dev Approve tokens for credit account. Restricted for adapters only
```

Note that the comment is incorrect as it can also be called by the CreditFacade.

While `CreditFacade.approve()` does check whether the token to be approved is supported, the adapters generally do not check this. `CreditManager.approveCreditAccount()` itself does not perform such a check on the given token address.

The lack of token validation may be used in an exploit.

Note the following should also be taken into account:

- CreditAccounts may receive other tokens e.g. as an airdrop. How should users be able to access/trade them?
- A token may have been "forbidden". Does this only apply to a new incoming asset or does this also block usage as an outgoing asset?

Code corrected:

Token being supported is now checked in `CreditManager.approveCreditAccount()`. This means that the token will be verified regardless of whether the call comes from the CreditFacade or an adapter.

On additional notes:

CreditFacade now has an `enableToken()` function which allows the Credit Account owner to enable any token and include it in the collateral computation, as long as this token is supported by the Credit Manager and is not forbidden. This can be used to handle airdropped tokens.

Whether the token is forbidden is only checked when a new token comes in and is being enabled (in CreditManager.checkAndEnableToken()). Whether an outgoing token is forbidden is not checked. This is deliberately done in order to allow positions in a forbidden token to be unwound after it was forbidden, by selling the token on Uniswap, closing/liquidating the account, etc.

The function annotation is outdated: `/// @dev Approve tokens for credit account. Restricted for adapters only.` The CreditFacade is also eligible to call this function.

7.12 Curve Registry

Correctness **Low** **Version 1** **Code Corrected**

The factory contract CurveLPFactory which deploys the curve price feeds has the address of the Curve registry hardcoded. Similarly, CurveV1_Base uses the hardcoded address.

According to the Curve Documentation of their registry contracts, the central source of truth in the Curve system is the address provider. That contract allows changing the registry through `set_address()` when the `id` parameter is set to zero. Currently, the oracle stores the registry as an immutable. Hence, in case the registry changes, the contract will utilize a wrong registry.

Code corrected:

The Curve Registry is no longer used either by the price feeds or CurveV1_Base and hence this issue no longer applies.

7.13 CurveV1 Adapters: TokenOut Might Not Be Enabled at CreditAccount

Security **Low** **Version 1** **Code Corrected**

The implementation of the CurveV1_2/3/4 adapters bears the risk that after a successful action, the incoming tokens might not be enabled at the CreditAccount.

Consider the following function:

```
function remove_liquidity(
    uint256 amount,
    uint256[N_COINS] memory min_amounts
) external virtual nonReentrant {
    address creditAccount = creditManager.getCreditAccountOrRevert(
        msg.sender
    ); // F:[ACV1_2-3]

    _enable_tokens(creditAccount, min_amounts);
    _executeFullCheck(creditAccount, msg.data); //F:[ACV1_2-5,6]
}
```

Parameter `min_amounts` serves as slippage protection. The adapter uses it to enable the incoming tokens using the internal `_enable_tokens` function:

```
function _enable_tokens(
    address creditAccount,
    uint256[N_COINS] memory amounts
) internal {
    if (amounts[0] > 1) {
        creditManager.checkAndEnableToken(creditAccount, token0); //F:[ACV1_2-5,6]
    }

    if (amounts[1] > 1) {
        creditManager.checkAndEnableToken(creditAccount, token1); //F:[ACV1_2-5,6]
    }
}
```

If the user didn't set the slippage protection (which shouldn't be done as it makes the transaction vulnerable to being sandwiched, resulting in worse exchange rates) the token might not be enabled in the credit account. This may remain undetected when the remaining assets at the credit account suffice to reach a health factor > 1. Closing such a credit account likely leaves those tokens behind and a later borrower who realizes this could collect them. Also, if such a credit account becomes unhealthy and is liquidated, a liquidator could collect the tokens.

Code corrected:

The function now enables all tokens of the pool, regardless of the `min_amounts` array. This is correct, since `remove_liquidity()` transfers tokens based on the current inventory of the pool, so there are only two scenarios in which it can return less than 2 tokens:

- the user burns a very small amount of the LP token;
- the pool is 100% unbalanced, which should not be practically achievable in Curve.

7.14 Duplicate Error Code Used

Design Low Version 1 Code Corrected

The library `Errors` contains error messages encoded as short strings to save on deployment cost and contract size. One of the error codes, "CFH", is used for two distinct errors, `CC_INCORRECT_TOKEN_CONTRACT` and `CM_TOKEN_IS_ALREADY_ADDED`, which prevents users from exactly determining the cause of the error.

Code corrected:

Text errors are being replaced with explicit Exceptions that are now being thrown on errors or constraint violations. In particular, errors in question were replaced with `IErrors.IncorrectTokenContractException` and `ICreditManagerV2Exceptions.TokenAlreadyAddedException`.

In the current version of the code the library Errors.sol is still imported and used by several system contracts, the duplicate error described above however has been corrected.

7.15 Incorrect Comment After Refactoring

Correctness Low Version 1 Code Corrected

A comment in `CreditManager.manageDebt` mentions that the function sometimes shifts `newBorrowedAmount`. This comment refers to a previous version of the code and isn't describing the current system.

Code corrected:

The comment has been removed

7.16 Incorrect Descriptions

Correctness Low Version 1 Specification Changed

PriceOracle:

- The function description of `addPriceFeed` in both the contract and the interface definition incorrectly mentions Eth

```
/// @param priceFeed Address of chainlink price feed token => Eth
```

In GearboxV2 the Chainlink pricefeed used is supposed to return a value in USD.

- the return value in `convertedToUSD()` is incorrectly described as:

```
/// @return Amount converted to tokenTo asset
```

- the description for parameter *token* should read *to* instead of *from* in the *convertFromUSD()*` function
- The description of `fastCheck()` is incorrect.
- Not all functions in the interface are annotated.

CreditFacade:

- The description of both functions `closeCreditAccount` and `liquidateCreditAccount` mention the outdated `sendAllAssets`.

CreditManager:

- `fastCollateralCheck` still mentions WETH instead of USD
- `closeCreditAccount` description mentions if sendAllAssets is true, this no longer exists.

Specification changed:

The aforementioned description issues have been rectified.

7.17 Outdated Function Description

Correctness Low Version 1 Specification Changed

There are frequent cases in which comments refer to previous functionality in the code which now has been changed. As an example, the description of function `closeCreditAccount` in both contracts, CreditFacade and CreditManager describe `sendAllAssets` which no longer exists. Similarly this applies to the function `liquidateCreditAccount` of the CreditFacade in which `skipTokenMask` allows this behavior now. .

Specification changed:

Function annotations have been brought up-to-date.

7.18 PriceOracle: Unused Timestamp

Design Low Version 1 Code Corrected

```
function _getPrice(address token) internal view returns (uint256) {
    require(
        priceFeeds[token] != address(0),
        Errors.PO_PRICE_FEED_DOESNT_EXIST
    );

    (
        ,
        //uint80 roundID,
        int256 price, //uint startedAt, , //uint80 answeredInRound
        ,
        uint256 timeStamp,

    ) = AggregatorV3Interface(priceFeeds[token]).latestRoundData();

    return uint256(price);
}
```

}

`latestRoundData()` returns several values, all unused values except timesTamp are dropped. The value for timeStamp is handled but remains unused.

Code corrected:

`PriceOracle.getPrice()` now uses `roundId`, `answer`, `updatedAt` and `answereInRound` to perform sanity checks on round data. The unused `startedAt` value is dropped.

7.19 Read-only Reentrancy

Design Low Version 1 Code Corrected

When integrating with Gearbox, developers should be aware of read-only reentrancy opportunities. Assume a credit account (CA) which is controlled by a protocol (P) integrating with Gearbox and holds WETH, and a malicious user (E). Assume now that at some point the account becomes liquidatable:

- E liquidates the account by calling `CreditFacade.liquidateCreditAccount` where the `to` address is a smart contract controlled by E and `convertWETH` is `true`.
- During closure, `CreditManager.closeCreditAccount` is called, which converts WETH to ETH and sends it to `to` as seen in the following snippet:

```
_transferAssetsTo(creditAccount, to, convertWETH, enabledTokensMask);
```

- At this point, the control of the execution flow is passed to the smart contract of `to` address.
- The smart contract makes a call to P which queries the state of CA. CA will seem like it holds less value than it actually used to at the beginning of the transaction. The reason is that its state hasn't been fully updated but part of its holdings has been sent to another address.
- Based on this intermediate state of the CA, P might proceed incorrectly and end up in an unexpected state.

Code corrected:

The line ` `delete creditAccounts[borrower]; // F:[CM-9]` ` was moved to the beginning of the function, right after the Credit Account for the borrower is first retrieved. This will make any calls to *CreditManager.getCreditAccountOrRevert()* in the middle of *closeCreditAccount* execution fail, since the record no longer exists in the mapping.

While third-party protocols that directly query the state through a saved CA address may still be vulnerable, we will advise all integrators to use *CreditManager.getCreditAccountOrRevert()* to retrieve the address dynamically, as a security best practice.

7.20 Redundant Event Emission

Design | Low | Version 1 | Code Corrected

When `CreditFacade._disableToken` is called, a `TokenDisabled` event is emitted even if the token was already disabled.

Code corrected:

CreditManager.disableToken() now returns whether the token was actually disabled. This is used in *CreditFacade._disableToken()* to emit the event conditionally.

7.21 Redundant Initialization

Design | Low | Version 1 | Code Corrected

In `CreditConfigurator.constructor` the following line can be found:

```
creditManager.upgradePriceOracle(address(creditManager.priceOracle())); // F:[CC-1]
```

This line upgrades the price oracle of the `CreditManager` with the same price oracle. Hence, this call is redundant.

Code corrected:

The line has been deleted.

7.22 Reentrancy Into CreditFacade

Security | Low | Version 1 | Acknowledged | Code Corrected

The new CreditFacade featuring the new multicall functionality allows executing multiple actions including calls to the adapters. A health check of the credit account is only done once after all calls have been executed, not in between calls. In between calls credit accounts may be in an unhealthy state. The internal multicall function of CreditFacade itself is not protected against reentrancy, nor are some functions of the CreditFacade using this multicall functionality. Reentrancy protection is present in the called adapter and during the execution of certain functions of the CreditManager. Note that the reentrancy protection used works per contract: Reentrancy into the specific contract is locked at the beginning of the function and the lock is released when the function call completes.

Aside from certain functions of the CreditFacade itself (which are handled differently), multicall allows calling any function on external contracts which are valid adapters.

Furthermore, note that attacks are limited as credit account cannot be returned in the very same block it has been borrowed.

Nevertheless, extra care should be taken especially as untrusted code can be reached via the adapters. It might be more cautious to prevent reentrancy into the CreditFacade as this is not intended to be done.

Code corrected and Acknowledged:

All remaining non-restricted CreditFacade functions have been covered with a nonReentrant modifier. This ensures that:

- At most one multicall can be performed within a single transaction (internal _multicall() can only be called from non-reentrant functions);
- Only one of debt-managing functions (addCollateral, increaseDebt and decreaseDebt) can be called externally within a single transaction (internal counterparts can be called multiple times within a multicall, barring flash loan protections).

7.23 Sanity Check of New Pricefeed

Design | Low | Version 1 | Code Corrected

`PriceOracle._addPriceFeed()` contains the following sanity check:

```
require(
    AggregatorV3Interface(priceFeed).decimals() == 8,
    Errors.PO_AGGREGATOR_DECIMALS_SHOULD_BE_8
); // F:[PO-2]
```

This check helps to ensure that the intended kind of pricefeeds returning a price with 8 decimal is passed, which USD-denominated Chainlink pricefeeds do.

The sanity check could be enhanced to check if the pricefeed actually implements the required functionality of the AggregatorV3Interface, notably whether function `latestRoundData` is implemented which is the function called by the PriceOracle to query the price.

Code corrected:

`_addPriceFeed()` now performs extensive sanity checks on the newly added feed and token:

Checks that neither feed nor token are zero addresses; Checks that the token is a contract; Checks that the price feed is a contract; Checks that the token implements `decimals()`; Checks that the feed implements `decimals()` and it is equal to 8; Checks that the feed implements `dependsOnAddress()`; Checks that the feed implements `skipPriceCheck()`; Checks that the feed implements `latestRoundData()` (and performs sanity checks on the answer if skipPriceCheck() == false);

7.24 Unused `allowedContractsSet`

Design | Low | Version 1 | Code Corrected

`EnumerableSet.AddressSet private allowedContractsSet;` defined in the CreditFacade is unused. The very same variable exists in the *CreditConfigurator* where it actually is used.

Code corrected:

Removed unused variable and corresponding getters.

7.25 YearnV2Adapter: Different Behavior of Functions

Design | Low | Version 1 | Code Corrected

Functions `transfer` and `transferFrom` revert with `Errors.NOT_IMPLEMENTED`. Function `approve()` however behaves differently simply returns `true`.

Code corrected:

`approve()`, `transfer()` and `transferFrom()` of the YearnV2Adapter now return `false` without doing anything.

8 Notes

We leverage this section to highlight further findings that are not necessarily issues. The mentioned topics serve to clarify or support the report, but do not require an immediate modification inside the project. Instead, they should raise awareness in order to improve the overall understanding.

8.1 Airdrops

Note Version 1

CreditAccounts may be eligible for airdrops, e.g. as they may have held a certain token when a snapshot was taken or as they may have interacted with a DeFi system a certain amount of times.

Users of a credit account must be aware that they lose participation in the airdrop when they return the credit account (close/liquidation).

At the moment when the information about an airdrop becomes public, this credit account may be in use or in the queue at the factory.

Depending on the value of the airdrop users may attempt to retrieve this credit account. The governance has the option to take out such accounts directly. Generally, airdrops can only be claimed by the credit account if this process can be triggered by a third party. Airdrops requiring the credit account to call a specific function generally won't work as no adapter supporting this exists.

8.2 Free Flashloans

Note Version 1

Gearbox prevents users from increasing and decreasing their debt during a multicall and thus taking a free flashloan. However, a user could still create a contract that executes two separate multicalls, one that includes a debt increase and one that includes a debt decrease. This way, a free loan is still possible. It is important to note that the amount to be borrowed during the loan is still limited by the checks performed when an amount is borrowed from the pool.

8.3 Renouncing Ownership

Note Version 1

In Gearbox, transfers of ownership take place in two steps. First, the previous owner defines the new owner (`pendingOwner`) and the new owner claims the ownership. The `Claimable` contract extends `Ownable` meaning that the old owner can renounce ownership. Users should note that ownership renounce is ignored if a pending owner has been already defined since `Claimable.claimOwnership` does not check if the ownership has been renounced before.

8.4 `_safeTokenTransfer` - Call to External Address

Note Version 1

When the boolean parameter `convertToETH` is set to true, WETH is unwrapped into Ether. This Ether is transferred to the recipient using a call, the gas amount passed is not restricted. At this point, the execution may reach untrusted code.

The function name "safeTokenTransfer" is due to the usage of OpenZeppelins SafeERC20 library. One must be careful to not misinterpret the function name and assume using this function is "safe" under all circumstances.