Goldfinch Protocol

Smart Contracts

Security Assessment

December 4th, 2020

By:
Camden Smallwood @ CertiK
camden.smallwood@certik.org

Sheraz Arshad @ CertiK
sheraz.arshad@certik.org

Alex Papageorgiou @ CertiK
alex.papageorgiou@certik.org

Disclaimer

CertiK reports are not, nor should be considered, an “endorsement” or “disapproval” of any particular project or team. These reports are not, nor should be considered, an indication of the economics or value of any “product” or “asset” created by any team or project that contracts CertiK to perform a security review.

CertiK Reports do not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor do they provide any indication of the technologies proprietors, business, business model or legal compliance.

CertiK Reports should not be used in any way to make decisions around investment or involvemeânt with any particular project. These reports in no way provide investment advice, nor should be leveraged as investment advice of any sort.

CertiK Reports represent an extensive auditing process intending to help our customers increase the quality of their code while reducing the high level of risk presented by cryptographic tokens and blockchain technology.

Blockchain technology and cryptographic assets present a high level of ongoing risk. CertiK’s position is that each company and individual are responsible for their own due diligence and continuous security. CertiK’s goal is to help reduce the attack vectors and the high level of variance associated with utilizing new and consistently changing technologies, and in no way claims any guarantee of security or functionality of the technology we agree to analyze.

What is a CertiK report?

- A document describing in detail an in depth analysis of a particular piece(s) of source code provided to CertiK by a Client.
- An organized collection of testing results, analysis and inferences made about the structure, implementation and overall best practices of a particular piece of source code.
- Representation that a Client of CertiK has indeed completed a round of auditing with the intention to increase the quality of the company/product's IT infrastructure and or source code.

Overview

Project Summary

Project Name	**Goldfinch Protocol**
Description	A lending protocol focused on enabling crypto holders to earn yield from real-world, off-chain borrowers. This is achieved by allowing anyone to contribute USDC to the pool. Governance-approved underwriters can then extend credit lines to individual borrowers, who can draw down capital from the pool for use off-chain.
Platform	Ethereum; Solidity, Yul
Codebase	GitHub Repository
Commits	pre-audit: f76d6cb099020d67ccc3baa216e6689e373914a9 post-audit: 2b49bf63742ee36d7410de49716d11864b9ed306

Audit Summary

Delivery Date	**December 4th, 2020**
Method of Audit	Static Analysis, Manual Review
Consultants Engaged	2
Timeline	November 16th, 2020 - December 4th, 2020

Vulnerability Summary

Total Issues	44
Total Critical	0
Total Major	0
Total Medium	0
Total Minor	4
Total Informational	40

Executive Summary

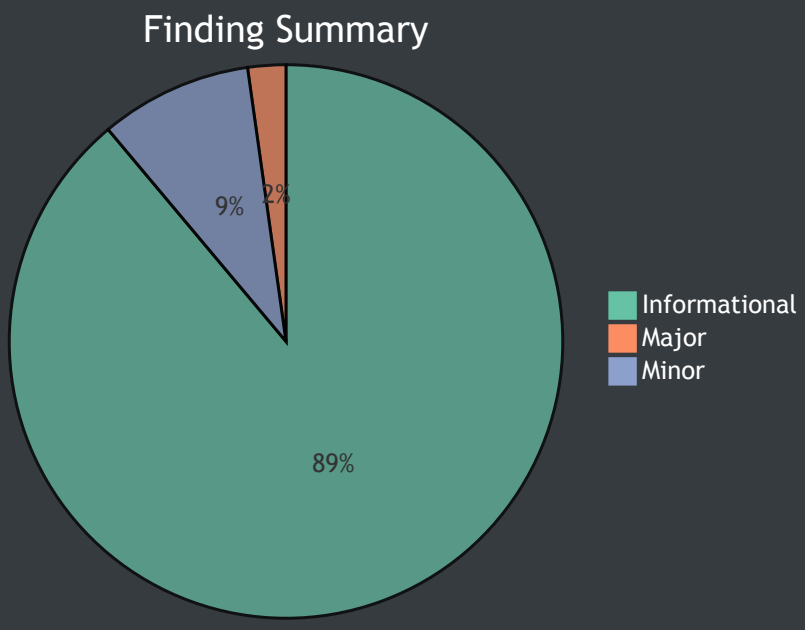
This report represents the results of CertiK's engagement with Goldfinch Protocol on their implementation of the Goldfinch Protocol smart contracts.

Our findings mainly refer to optimizations and Solidity coding standards. Although two minor exhibits remain unresolved, the overall security of the codebase can be deemed as high and the identified issues pose no threat to the safety of the contract deployment.

Files In Scope

ID	Contract	Location
ACC	Accountant.sol	contracts/protocol/Accountant.sol
BUP	BaseUpgradeablePausable.sol	contracts/protocol/BaseUpgradeablePausable.sol
CDK	CreditDesk.sol	contracts/protocol/CreditDesk.sol
CLE	CreditLine.sol	contracts/protocol/CreditLine.sol
CHR	ConfigHelper.sol	contracts/protocol/ConfigHelper.sol
COS	ConfigOptions.sol	contracts/protocol/ConfigOptions.sol
CLF	CreditLineFactory.sol	contracts/protocol/CreditLineFactory.sol
FID	Token.sol	contracts/protocol/Token.sol
GCG	GoldfinchConfig.sol	contracts/protocol/GoldfinchConfig.sol
IFU	IToken.sol	contracts/interfaces/IToken.sol
IPL	IPool.sol	contracts/interfaces/IPool.sol
ICD	ICreditDesk.sol	contracts/interfaces/ICreditDesk.sol
IER	IERC20withDec.sol	contracts/interfaces/IERC20withDec.sol
GPO	Pool.sol	contracts/protocol/Pool.sol
PPE	PauserPausable.sol	contracts/protocol/PauserPausable.sol

Findings



ID	Title	Type	Severity	Resolved
GPO-01	Unlocked Compiler Version	Language Specific	Informational	✓
GPO-02	`require` call can be subsituted with a `modifier`	Coding Style	Informational	✓
GPO-04	Potential for addition overflow in `collectInterestRepayment`	Mathematical Operations	Minor	✓
GPO-05	Result of call to `doUSDCTransfer` is ignored	Logical Issue	Minor	✓

GPO-06	Inefficient `sharePrice` state variable access	Gas Optimization	Informational	✓
GPO-07	Inefficient usage of local variable as constant	Gas Optimization	Informational	✓
CLE-01	Unlocked Compiler Version	Language Specific	Informational	✓
CDK-01	Unlocked Compiler Version	Language Specific	Informational	✓
CDK-02	Inefficient local variable declaration	Gas Optimization	Informational	✓
CDK-03	ABI encoding with explicit function signature	Volatile Code	Informational	✓
CDK-04	Unnecessary casting to `address`	Gas Optimization	Informational	✓
CDK-05	Logic error in `assessCreditLine` early exit	Volatile Code	Minor	!
CDK-06	Inefficient Greater-Than Comparison w/ Zero	Gas Optimization	Informational	✓
CDK-07	`less-than-or-equal to zero` comparison with `uint256` variable	Mathematical Operations	Informational	✓
CDK-08	Redundant Variable Initialization	Coding Style	Informational	✓
CDK-09	Redundant Variable Initialization	Coding Style	Informational	✓
CDK-10	Inefficient `storage` read	Gas Optimization	Informational	✓
CDK-11	Explicitly returning a local variable	Gas Optimization	Informational	!
CDK-12	`require` call can be substituted with a `modifier`	Coding Style	Informational	!
CDK-13	`require` call can be substituted with a `modifier`	Coding Style	Informational	✓

CDK-14	Typo in the comment	Coding Style	Informational	✓
CDK-15	Inefficient code	Gas Optimization	Informational	✓
CDK-16	An `underwriter` with `0` governance limit can create credit line with `0` limit	Control Flow	Informational	!
CDK-17	Credit lines cannot be removed from credit desk	Volatile Code	Minor	!
BUP-01	Unlocked Compiler Version	Language Specific	Informational	✓
CHR-01	Unlocked Compiler Version	Language Specific	Informational	✓
COS-01	Unlocked Compiler Version	Language Specific	Informational	✓
CLF-01	Unlocked Compiler Version	Language Specific	Informational	✓
CLF-02	Function Visibility Optimization	Gas Optimization	Informational	✓
FID-01	Unlocked Compiler Version	Language Specific	Informational	✓
FID-02	Function Visibility Optimization	Gas Optimization	Informational	✓
FID-03	Local variable can be substituted with state constant	Gas Optimization	Informational	✓
ACC-01	Unlocked Compiler Version	Language Specific	Informational	✓
ACC-02	The `else` block can be removed	Gas Optimization	Informational	!
ACC-03	Inefficient Greater-Than Comparison w/ Zero	Gas Optimization	Informational	!
GCG-01	Unlocked Compiler Version	Language Specific	Informational	✓

GCG-02	Redundant casting of `uint256` type to `uint256`	Gas Optimization	Informational	✓
GCG-03	Inefficient local variable declaration	Gas Optimization	Informational	✓
PPE-01	Unlocked Compiler Version	Language Specific	Informational	✓
PPE-02	`require` call can be substituted with a `modifier`	Coding Style	Informational	✓
ICD-01	Unlocked Compiler Version	Language Specific	Informational	✓
IER-01	Unlocked Compiler Version	Language Specific	Informational	✓
IFU-01	Unlocked Compiler Version	Language Specific	Informational	✓
IPL-01	Unlocked Compiler Version	Language Specific	Informational	✓

GPO-01: Unlocked Compiler Version

Type	Severity	Location
Language Specific	Informational	Pool.sol L3

Description:

The contract has unlocked compiler version. An unlocked compiler version in the source code of the contract permits the user to compile it at or above a particular version. This, in turn, leads to differences in the generated bytecode between compilations due to differing compiler version numbers. This can lead to an ambiguity when debugging as compiler specific bugs may occur in the codebase that would be hard to identify over a span of multiple compiler versions rather than a specific one.

Recommendation:

We advise that the compiler version is instead locked at the lowest version possible that the contract can be compiled at. For example, for version `v0.6.8` the contract should contain the following line:

```
pragma solidity 0.6.8;
```

Alleviation:

The development team opted to consider our references and locked the compiler to version `0.6.12`.

GPO-02: `require` call can be subsituted with a `modifier`

Type	Severity	Location
Coding Style	Informational	Pool.sol L53, L71, L243

Description:

The `require` calls on the aforementioned lines can be subsituted with a `modifier` to increase the legibility of the code.

Recommendation:

We advise to introduce a `modifier` and use it in place of the `require` statements on the aforementioned lines.

```
modifier onlyTransactionWithinLimit(uint256 amount) {
   require(transactionWithinLimit(amount), "Amount is over the per-transaction limit");
   _;
}
```

The usage of the modifier is shown in the following code snippet.

```
function fn_name(uint256 amount) onlyTransactionWithinLimit(amount) {...}
```

Alleviation:

The development team opted to consider our references and implemented the proposed `modifier` and applied it throughout the contract.

GPO-04: Potential for addition overflow in `collectInterestRepayment`

Type	Severity	Location
Mathematical Operations	Minor	Pool.sol L107

Description:

The `collectInterestRepayment` function in the `Pool` contract performs a primitive addition `sharePrice` and `increment` without checking either value for potential overflow:

```
    sharePrice = sharePrice + increment;
```

Recommendation:

We advise to perform safe addition by using `add` function from `SafeMath` library:

```
    sharePrice = sharePrice.add(increment);
```

Alleviation:

The development team opted to consider our references and used the `add` function from `SafeMath` library.

GPO-05: Result of call to `doUSDCTransfer` is ignored

Type	Severity	Location
Logical Issue	Minor	Pool.sol L59, L83, L109, L128

Description:

The function `doUSDCTransfer` returns a boolean value when it is called. The aforementioned lines call this function but do not check the returned value of the function call to determine the successful status of the function's exectution.

As the contract will evolve in future to work with different token contracts other than USDC, there will be a problem with the implementation of `doUSDCTransfer`. Many token implementations do not return anything upon the execution of `transferFrom` i.e. Tether (USDT) leading to unexpected halts in code execution.

Recommendation:

We recommend to change the implementation of `doUSDCTransfer` where it does not return anything and the `safeTransferFrom` from `SafeERC20` is used to perform the transfer operation instead of `transferFrom` which invokes the function safely in all circumstances whether the function returns value or not.

```
function doUSDCTransfer(
   address from,
   address to,
   uint256 amount
) {
   ...
   usdc.safeTransferFrom(from, to, amount);
   ...
}
```

Alleviation:

The development team opted to consider our references, stored the returned value of each linked function invocation and added a `require` statement to ensure that the returned value is acceptable.

GPO-06: Inefficient `sharePrice` state variable access

Type	Severity	Location
Gas Optimization	Informational	Pool.sol L212

Description:

The `assetsMatchLiabilities` function in the `Pool` contract retrieves the value of the `sharePrice` state variable using `config.getPool().sharePrice()` , yet the `sharePrice` state variable is directly accessible through the `Pool` contract itself. This implementation results in increased gas cost for the operation.

Recommendation:

We advise to access the `sharePrice` state variable through the `Pool` contract directly in order to improve the legibility of the codebase and save on the overall cost of gas:

```
uint256 liabilities =
config.getToken().totalSupply().mul(sharePrice).div(tokenMantissa());
```

Alleviation:

The development team opted to consider our references and optimized the linked statement as proposed.

GPO-07: Inefficient usage of local variable as constant

Type	Severity	Location
Gas Optimization	Informational	Pool.sol L216

Description:

The `assetsMatchLiabilities` function in the `Pool` contract declares a local `threshold` variable with a constant value of `1e6`, but never modifies it which is inefficient:

```
uint256 threshold = 1e6;
```

Recommendation:

We advise to remove the local `threshold` variable and re-declare it as a state `constant` in order to save on the overall cost of gas:

```
uint256 constant THRESHOLD = 1e6;
```

```
if (_assets >= liabilitiesInDollars) {
  return _assets.sub(liabilitiesInDollars) <= THRESHOLD;
} else {
  return liabilitiesInDollars.sub(_assets) <= THRESHOLD;
}
```

Alleviation:

The development team opted to consider our references and introduced the `constant` variable `ASSET_LIABILITY_MATCH_THRESHOLD`.

CLE-01: Unlocked Compiler Version

Type	Severity	Location
Language Specific	Informational	CreditLine.sol L3

Description:

The contract has unlocked compiler version. An unlocked compiler version in the source code of the contract permits the user to compile it at or above a particular version. This, in turn, leads to differences in the generated bytecode between compilations due to differing compiler version numbers. This can lead to an ambiguity when debugging as compiler specific bugs may occur in the codebase that would be hard to identify over a span of multiple compiler versions rather than a specific one.

Recommendation:

We advise that the compiler version is instead locked at the lowest version possible that the contract can be compiled at. For example, for version `v0.6.8` the contract should contain the following line:

```
pragma solidity 0.6.8;
```

Alleviation:

The development team opted to consider our references and locked the compiler to version `0.6.12`.

CDK-01: Unlocked Compiler Version

Type	Severity	Location
Language Specific	Informational	CreditDesk.sol L3

Description:

The contract has unlocked compiler version. An unlocked compiler version in the source code of the contract permits the user to compile it at or above a particular version. This, in turn, leads to differences in the generated bytecode between compilations due to differing compiler version numbers. This can lead to an ambiguity when debugging as compiler specific bugs may occur in the codebase that would be hard to identify over a span of multiple compiler versions rather than a specific one.

Recommendation:

We advise that the compiler version is instead locked at the lowest version possible that the contract can be compiled at. For example, for version `v0.6.8` the contract should contain the following line:

```
pragma solidity 0.6.8;
```

Alleviation:

The development team opted to consider our references and locked the compiler to version `0.6.12`.

CDK-02: Inefficient local variable declaration

Type	Severity	Location
Gas Optimization	Informational	CreditDesk.sol L74

Description:

The `setUnderwriterGovernanceLimit` function in the `CreditDesk` contract declares a local storage pointer variable, which is inefficient as its initialization part is used only used once in the code:

```
    Underwriter storage underwriter = underwriters[underwriterAddress];
```

Recommendation:

We advise to use to directly use the initialization part of the local variable on `L76` to save gas cost associated with declaring local variable.

```
    underwriters[underwriterAddress].governanceLimit = limit;
```

Alleviation:

The development team opted to consider our references and modified the `setUnderwriterGovernanceLimit` function as proposed.

CDK-03: ABI encoding with explicit function signature

Type	Severity	Location
Volatile Code	Informational	CreditDesk.sol L111

Description:

The `createCreditLine` function in the `CreditDesk` contract ABI encodes the data for the `initialize` function in the `CreditDesk` by specifying the signature explicitly, which has the potential to break in the event of a refactor:

```
bytes memory arguments = abi.encodeWithSignature(
  "initialize(address,address,address,uint256,uint256,uint256,uint256,uint256)",
   ...
);
```

Recommendation:

We recommend to use the actual signature by importing the target contract to ensure that an interface change won't break the functionality of this contract.

Alleviation:

The development team opted to consider our references and modified the linked code segment as proposed.

CDK-04: Unnecessary casting to `address`

Type	Severity	Location
Gas Optimization	Informational	CreditDesk.sol L125-L128

Description:

The `cl` local variable is casted to `address` and used on the aforementioned lines yet the `address` representation of `cl` is already accessible in the code on `L122` with variable `clAddress`.

Recommendation:

We recommend to use the local variable `clAddress` on the aforementioned lines instead of casting `cl` to `address` as the use of `clAddress` would be gas efficient.

Alleviation:

The development team opted to consider our references and modified the linked code segment as proposed.

CDK-05: Logic error in `assessCreditLine` early exit

Type	Severity	Location
Volatile Code	Minor	CreditDesk.sol L210

Description:

For CDK-05, we noticed the following piece of code does not align with the comment describing its functionality, as it checks to see if both a full period has passed and if the credit line is late using an AND operation, as opposed to either/or using an OR operation:

```
// Do not assess until a full period has elapsed or past due
if (blockNumber() < cl.nextDueBlock() && !isLate(cl)) {
    return;
}
```

Recommendation:

We advise the team to change from an AND operation to an OR operation.

Alleviation:

The development team has acknowledged this exhibit but decided to not apply its remediation in the current version of the codebase.

CDK-06: Inefficient Greater-Than Comparison w/ Zero

Type	Severity	Location
Gas Optimization	Informational	CreditDesk.sol L367

Description:

The linked greater-than comparisons with zero compare variables that are restrained to the non-negative integer range, meaning that the comparator can be changed to an inequality one which is more gas efficient.

Recommendation:

We advise that the above paradigm is applied to the linked greater-than statements.

Alleviation:

The development team has acknowledged this exhibit but decided to use `CreditLine`'s variable `lastFullPaymentBlock` instead of `Accountant`'s function `calculateWritedownFor` to the conditional, hence remediating this exhibit.

CDK-07: `less-than-or-equal to zero` comparison with `uint256` variable

Type	Severity	Location
Mathematical Operations	Informational	CreditDesk.sol L410, L425

Description:

The aforementioned line perform `less-than-or-equal to zero` comparison with `cl.balance()`. The function `cl.balance()` returns a `uint256` value which can never have a value less than zero.

Recommendation:

We recommend to change the comparison from `less-than-or-equal to zero` to `is-equal to zero` to increase the legibility of the codebase as `uint256` can never have a value less than zero.

Alleviation:

The development team opted to consider our references and used equality comparison to the linked conditionals.

CDK-08: Redundant Variable Initialization

Type	Severity	Location
Coding Style	Informational	CreditDesk.sol L462

Description:

All variable types within Solidity are initialized to their default "empty" value, which is usually their zeroed out representation. Particularly:

- `uint / int` : All `uint` and `int` variable types are initialized at `0`
- `address` : All `address` types are initialized to `address(0)`
- `byte` : All `byte` types are initialized to their `byte(0)` representation
- `bool` : All `bool` types are initialized to `false`
- `ContractType` : All contract types (i.e. for a given `contract ERC20 {}` its contract type is `ERC20`) are initialized to their zeroed out address (i.e. for a given `contract ERC20 {}` its default value is `ERC20(address(0))`)
- `struct` : All `struct` types are initialized with all their members zeroed out according to this table

Recommendation:

We advise that the linked initialization statements are removed from the codebase to increase legibility.

Alleviation:

The development team opted to consider our references and removed the redundant variable initialization.

CDK-09: Redundant Variable Initialization

Type	Severity	Location
Coding Style	Informational	CreditDesk.sol L289

Description:

All variable types within Solidity are initialized to their default "empty" value, which is usually their zeroed out representation. Particularly:

- `uint / int` : All `uint` and `int` variable types are initialized at `0`
- `address` : All `address` types are initialized to `address(0)`
- `byte` : All `byte` types are initialized to their `byte(0)` representation
- `bool` : All `bool` types are initialized to `false`
- `ContractType` : All contract types (i.e. for a given `contract ERC20 {}` its contract type is `ERC20`) are initialized to their zeroed out address (i.e. for a given `contract ERC20 {}` its default value is `ERC20(address(0))`)
- `struct` : All `struct` types are initialized with all their members zeroed out according to this table

Recommendation:

We advise that the linked initialization statements are removed from the codebase to increase legibility.

Alleviation:

The development team opted to consider our references and removed the redundant variable initialization.

CDK-10: Inefficient `storage` read

Type	Severity	Location
Gas Optimization	Informational	CreditDesk.sol L463

Description:

The `for` loop on the aforementioned line accesses the same storage slot to read array length for each iteration of the loop which consumes unnecessary gas that can be saved by storing the value in a local variable.

Recommendation:

We recommend to introduce a local variable and store the length in it which is read on each iteration of the `for` loop.

```
    uint256 length = underwriter.creditLines.length;

    for (uint256 i = 0; i < length; i++) {...}
```

Alleviation:

The development team opted to consider our references, stored the `underwriter.creditLines.length` value to a local variables and used this variable at the loop conditional.

CDK-11: Explicitly returning a local variable

Type	Severity	Location
Gas Optimization	Informational	CreditDesk.sol L461

Description:

The function on the aforementioned line explicitly return a local variable which increases overall cost of gas.

Recommendation:

Since named return variables can be declared in the signature of a function, consider refactoring to remove the local variable declaration and explicit return statement in order to reduce the overall cost of gas.

Alleviation:

The development team has acknowledged this exhibit but decided to not apply its remediation in the current version of the codebase.

CDK-12: `require` call can be substituted with a `modifier`

Type	Severity	Location
Coding Style	Informational	CreditDesk.sol L158, L256

Description:

The `require` calls on the aforementioned lines can be substituted with a modifier to increase the legibility of the codebase.

Recommendation:

We advise to use the modifier in place of the `require` calls on the aforementioned lines.

```
modifier onlyWithinTransactionLimit(uint256 amount) {
   require(withinTransactionLimit(amount), "Amount is over the per-transaction limit");
   _;
}
```

The usage is shown in the following code snippet.

```
function fnName(uint256 amount) onlyWithinTransactionLimit(amount) {...}
```

Alleviation:

The development team has acknowledged this exhibit but decided to not apply its remediation in the current version of the codebase.

CDK-13: `require` call can be substituted with a `modifier`

Type	Severity	Location
Coding Style	Informational	CreditDesk.sol L207, L192, L155

Description:

The `require` calls on the aforementioned lines can be substituted with modifier to increase the legibility of the codebase.

Recommendation:

We recommend to use modifier in place of the `require` calls on the aforementioned lines.

```
modifier onlyValidCreditLine(address creditLineAddress) {
   require(creditLines[creditLineAddress] != address(0), "Unknown credit line");
   _;
}
```

Alleviation:

The development team opted to consider our references and implemented the proposed `modifier` and applied it throughout the contract.

CDK-14: Typo in the comment

Type	Severity	Location
Coding Style	Informational	CreditDesk.sol L217

Description:

The aforementioned line has incorrect spelling for the word `assess`.

Recommendation:

We advise the rectify the spellings for the discussed word on the aforementioned line.

Alleviation:

The development team opted to consider our references and updated the linked comments.

CDK-15: Inefficient code

Type	Severity	Location
Gas Optimization	Informational	CreditDesk.sol L289-L301

Description:

The code block on the aforementioned lines uses a local variable `paymentApplied` to determine if the event `PaymentApplied` is needed to fired or not. The gas cost of executing this function can be saved by simpling emitting the event in the bodies of both `if` statements on lines `L290` and `L294`.

Recommendation:

We recommend to directly emit the event inside the body of `if` statements instead of using a local variable.

```
if (interestPayment > 0) {
      emit PaymentApplied(cl.borrower(), address(cl), interestPayment,
principalPayment, paymentRemaining);
      config.getPool().collectInterestRepayment(address(cl), interestPayment);
}
    if (principalPayment > 0) {
      emit PaymentApplied(cl.borrower(), address(cl), interestPayment,
principalPayment, paymentRemaining);
      config.getPool().collectPrincipalRepayment(address(cl), principalPayment);
}
```

Alleviation:

The development team opted to consider our references and modified the linked code segment as proposed.

CDK-16: An `underwriter` with `0` governance limit can create credit line with `0` limit

Type	Severity	Location
Control Flow	Informational	CreditDesk.sol L98

Description:

The function `createCreditLine` allows an `underwriter` with `0` governance limit to create a credit line with `0` limit. Although, it does not result in any loss of funds but an optimal code flow should not allow an `underwriter` with `0` governance limit to create credit line in any circumstances.

Recommendation:

We advise to add a check in the `createCreditLine` function to revert if it is called by an `underwriter` with `0` governance limit.

```
require(underwriter.governanceLimit != 0, "underwriter does not have governance limit");
```

Alleviation:

The development team has acknowledged this exhibit but decided to not apply its remediation in the current version of the codebase.

CDK-17: Credit lines cannot be removed from credit desk

Type	Severity	Location
Control Flow	Minor	CreditDesk.sol

Description:

The `creditLines` fields in the `Underwriter` and `Borrower` structures in the `CreditDesk` contract are only ever pushed to, and there is no way to remove from them, leading to an ever-increasing array. As a result, the `getCreditCurrentlyExtended` function and other pieces of the `CreditDesk` contract may be non-scalable.

Recommendation:

Consider incorporating a method to remove credit lines for the credit desk implementation in order to handle the case when a credit line is no longer valid in order to clear up space in the `creditLines` arrays in the `Underwriter` and `Borrower` structures, allowing the `CreditDesk` contract to be scaled further.

Alleviation:

The development team has acknowledged this exhibit but decided to not apply its remediation in the current version of the codebase.

BUP-01: Unlocked Compiler Version

Type	Severity	Location
Language Specific	Informational	BaseUpgradeablePausable.sol L3

Description:

The contract has unlocked compiler version. An unlocked compiler version in the source code of the contract permits the user to compile it at or above a particular version. This, in turn, leads to differences in the generated bytecode between compilations due to differing compiler version numbers. This can lead to an ambiguity when debugging as compiler specific bugs may occur in the codebase that would be hard to identify over a span of multiple compiler versions rather than a specific one.

Recommendation:

We advise that the compiler version is instead locked at the lowest version possible that the contract can be compiled at. For example, for version `v0.6.8` the contract should contain the following line:

```
pragma solidity 0.6.8;
```

Alleviation:

The development team opted to consider our references and locked the compiler to version `0.6.12`.

CHR-01: Unlocked Compiler Version

Type	Severity	Location
Language Specific	Informational	ConfigHelper.sol L3

Description:

The contract has unlocked compiler version. An unlocked compiler version in the source code of the contract permits the user to compile it at or above a particular version. This, in turn, leads to differences in the generated bytecode between compilations due to differing compiler version numbers. This can lead to an ambiguity when debugging as compiler specific bugs may occur in the codebase that would be hard to identify over a span of multiple compiler versions rather than a specific one.

Recommendation:

We advise that the compiler version is instead locked at the lowest version possible that the contract can be compiled at. For example, for version `v0.6.8` the contract should contain the following line:

```
pragma solidity 0.6.8;
```

Alleviation:

The development team opted to consider our references and locked the compiler to version `0.6.12`.

COS-01: Unlocked Compiler Version

Type	Severity	Location
Language Specific	Informational	ConfigOptions.sol L3

Description:

The contract has unlocked compiler version. An unlocked compiler version in the source code of the contract permits the user to compile it at or above a particular version. This, in turn, leads to differences in the generated bytecode between compilations due to differing compiler version numbers. This can lead to an ambiguity when debugging as compiler specific bugs may occur in the codebase that would be hard to identify over a span of multiple compiler versions rather than a specific one.

Recommendation:

We advise that the compiler version is instead locked at the lowest version possible that the contract can be compiled at. For example, for version `v0.6.8` the contract should contain the following line:

```
pragma solidity 0.6.8;
```

Alleviation:

The development team opted to consider our references and locked the compiler to version `0.6.12`.

CLF-01: Unlocked Compiler Version

Type	Severity	Location
Language Specific	Informational	CreditLineFactory.sol L3

Description:

The contract has unlocked compiler version. An unlocked compiler version in the source code of the contract permits the user to compile it at or above a particular version. This, in turn, leads to differences in the generated bytecode between compilations due to differing compiler version numbers. This can lead to an ambiguity when debugging as compiler specific bugs may occur in the codebase that would be hard to identify over a span of multiple compiler versions rather than a specific one.

Recommendation:

We advise that the compiler version is instead locked at the lowest version possible that the contract can be compiled at. For example, for version `v0.6.8` the contract should contain the following line:

```
pragma solidity 0.6.8;
```

Alleviation:

The development team opted to consider our references and locked the compiler to version `0.6.12`.

CLF-02: Function Visibility Optimization

Type	Severity	Location
Gas Optimization	Informational	CreditLineFactory.sol L23

Description:

The linked function is declared as `public`, contains array function arguments and is not invoked in any of the contract's contained within the project's scope.

Recommendation:

We advise that the functions' visibility specifiers are set to `external` and the array-based arguments change their data location from `memory` to `calldata`, optimizing the gas cost of the function.

Alleviation:

The development team opted to consider our references and modified the `createCreditLine` function as proposed.

FID-01: Unlocked Compiler Version

Type	Severity	Location
Language Specific	Informational	Token.sol L2

Description:

The contract has unlocked compiler version. An unlocked compiler version in the source code of the contract permits the user to compile it at or above a particular version. This, in turn, leads to differences in the generated bytecode between compilations due to differing compiler version numbers. This can lead to an ambiguity when debugging as compiler specific bugs may occur in the codebase that would be hard to identify over a span of multiple compiler versions rather than a specific one.

Recommendation:

We advise that the compiler version is instead locked at the lowest version possible that the contract can be compiled at. For example, for version `v0.6.8` the contract should contain the following line:

```
pragma solidity 0.6.8;
```

Alleviation:

The development team opted to consider our references and locked the compiler to version `0.6.12`.

FID-02: Function Visibility Optimization

Type	Severity	Location
Gas Optimization	Informational	Token.sol L27

Description:

The linked function is declared as `public`, contains array function arguments and is not invoked in any of the contract's contained within the project's scope.

Recommendation:

We advise that the functions' visibility specifiers are set to `public` and the array-based arguments change their data location from `memory` to `calldata`, optimizing the gas cost of the function.

Alleviation:

The development team opted to consider our references and modified the `__initialize__` function as proposed.

FID-03: Local variable can be substituted with state constant

Type	Severity	Location
Gas Optimization	Informational	Token.sol L91, L105

Description:

The local variables on the aforementioned lines declared with a literal value can be substituted with a state level `constant` which will be cheaper in gas cost.

Recommendation:

We advise to use a state `constant` instead of local variable on the aforementioned line to save gas cost.

```
uint256 constant THRESHOLD = 1e6;
```

```
if (_assets >= liabilitiesInDollars) {
      return _assets.sub(liabilitiesInDollars) <= THRESHOLD;
    } else {
      return liabilitiesInDollars.sub(_assets) <= THRESHOLD;
}
```

Alleviation:

The development team opted to consider our references and introduced the `constant` variable `ASSET_LIABILITY_MATCH_THRESHOLD`.

ACC-01: Unlocked Compiler Version

Type	Severity	Location
Language Specific	Informational	Accountant.sol L3

Description:

The contract has unlocked compiler version. An unlocked compiler version in the source code of the contract permits the user to compile it at or above a particular version. This, in turn, leads to differences in the generated bytecode between compilations due to differing compiler version numbers. This can lead to an ambiguity when debugging as compiler specific bugs may occur in the codebase that would be hard to identify over a span of multiple compiler versions rather than a specific one.

Recommendation:

We advise that the compiler version is instead locked at the lowest version possible that the contract can be compiled at. For example, for version `v0.6.8` the contract should contain the following line:

```
pragma solidity 0.6.8;
```

Alleviation:

The development team opted to consider our references and locked the compiler to version `0.6.12`.

ACC-02: The `else` block can be removed

Type	Severity	Location
Gas Optimization	Informational	Accountant.sol L49-L51

Description:

The `else` block on the aforementioned lines return `0` and it can be removed as the return type of the function is `uint256` and it will default to `0`.

Recommendation:

We recommend to remove the `else` block as the return value type `uint256` defaults to `0`.

Alleviation:

The development team has acknowledged this exhibit but decided to not apply its remediation in the current version of the codebase.

ACC-03: Inefficient Greater-Than Comparison w/ Zero

Type	Severity	Location
Gas Optimization	Informational	Accountant.sol L141

Description:

The linked greater-than comparisons with zero compare variables that are restrained to the non-negative integer range, meaning that the comparator can be changed to an inequality one which is more gas efficient.

Recommendation:

We advise that the above paradigm is applied to the linked greater-than statements.

Alleviation:

The development team has acknowledged this exhibit but decided to not apply its remediation in the current version of the codebase.

GCG-01: Unlocked Compiler Version

Type	Severity	Location
Language Specific	Informational	GoldfinchConfig.sol L3

Description:

The contract has unlocked compiler version. An unlocked compiler version in the source code of the contract permits the user to compile it at or above a particular version. This, in turn, leads to differences in the generated bytecode between compilations due to differing compiler version numbers. This can lead to an ambiguity when debugging as compiler specific bugs may occur in the codebase that would be hard to identify over a span of multiple compiler versions rather than a specific one.

Recommendation:

We advise that the compiler version is instead locked at the lowest version possible that the contract can be compiled at. For example, for version `v0.6.8` the contract should contain the following line:

```
pragma solidity 0.6.8;
```

Alleviation:

The development team opted to consider our references and locked the compiler to version `0.6.12`.

GCG-02: Redundant casting of `uint256` type to `uint256`

Type	Severity	Location
Gas Optimization	Informational	GoldfinchConfig.sol L68

Description:

The aforementioned line perform redundant casting of variable of type `uint256` to `uint256`.

Recommendation:

We advise to remove the explicit casting of variable `number` to `uint256` as the variable's type is already a `uint256`.

Alleviation:

The development team opted to consider our references and removed the redundant variable casting.

GCG-03: Inefficient local variable declaration

Type	Severity	Location
Gas Optimization	Informational	GoldfinchConfig.sol L30, L36, L43, L50

Description:

The local variables declarations on the aforementioned lines are inefficient as their initialization parts are only used once in the code.

Recommendation:

We recommend to use the initialization parts of the variable declarations directly in place of the variable to save gas cost associated with local variable declarations.

Alleviation:

The development team opted to consider our references and modified the linked statements as proposed.

PPE-01: Unlocked Compiler Version

Type	Severity	Location
Language Specific	Informational	PauserPausable.sol L2

Description:

The contract has unlocked compiler version. An unlocked compiler version in the source code of the contract permits the user to compile it at or above a particular version. This, in turn, leads to differences in the generated bytecode between compilations due to differing compiler version numbers. This can lead to an ambiguity when debugging as compiler specific bugs may occur in the codebase that would be hard to identify over a span of multiple compiler versions rather than a specific one.

Recommendation:

We advise that the compiler version is instead locked at the lowest version possible that the contract can be compiled at. For example, for version `v0.6.8` the contract should contain the following line:

```
pragma solidity 0.6.8;
```

Alleviation:

The development team opted to consider our references and locked the compiler to version `0.6.12`.

PPE-02: `require` call can be substituted with a `modifier`

Type	Severity	Location
Coding Style	Informational	PauserPausable.sol L34, L48

Description:

The `require` calls on the aforementioned lines can be replaced with a `modifier` to increase the legibility of the codebase.

Recommendation:

We recommend to use the `modifier` in place of `require` calls on the aforementioned lines.

```
modifier onlyPauserRole() {
   require(hasRole(PAUSER_ROLE, _msgSender()), "Must have pauser role to unpause");
   _;
}
```

The usage of `modifier` is shown in the following code snippet.

```
function pause() public onlyPauserRole {...}
function unpause() public onlyPauserRole {...}
```

Alleviation:

The development team opted to consider our references and implemented the proposed `modifier` and applied it throughout the contract.

ICD-01: Unlocked Compiler Version

Type	Severity	Location
Language Specific	Informational	ICreditDesk.sol L3

Description:

The contract has unlocked compiler version. An unlocked compiler version in the source code of the contract permits the user to compile it at or above a particular version. This, in turn, leads to differences in the generated bytecode between compilations due to differing compiler version numbers. This can lead to an ambiguity when debugging as compiler specific bugs may occur in the codebase that would be hard to identify over a span of multiple compiler versions rather than a specific one.

Recommendation:

We advise that the compiler version is instead locked at the lowest version possible that the contract can be compiled at. For example, for version `v0.6.8` the contract should contain the following line:

```
pragma solidity 0.6.8;
```

Alleviation:

The development team opted to consider our references and locked the compiler to version `0.6.12`.

IER-01: Unlocked Compiler Version

Type	Severity	Location
Language Specific	Informational	IERC20withDec.sol L3

Description:

The contract has unlocked compiler version. An unlocked compiler version in the source code of the contract permits the user to compile it at or above a particular version. This, in turn, leads to differences in the generated bytecode between compilations due to differing compiler version numbers. This can lead to an ambiguity when debugging as compiler specific bugs may occur in the codebase that would be hard to identify over a span of multiple compiler versions rather than a specific one.

Recommendation:

We advise that the compiler version is instead locked at the lowest version possible that the contract can be compiled at. For example, for version `v0.6.8` the contract should contain the following line:

```
pragma solidity 0.6.8;
```

Alleviation:

The development team opted to consider our references and locked the compiler to version `0.6.12`.

IFU-01: Unlocked Compiler Version

Type	Severity	Location
Language Specific	Informational	IToken.sol L3

Description:

The contract has unlocked compiler version. An unlocked compiler version in the source code of the contract permits the user to compile it at or above a particular version. This, in turn, leads to differences in the generated bytecode between compilations due to differing compiler version numbers. This can lead to an ambiguity when debugging as compiler specific bugs may occur in the codebase that would be hard to identify over a span of multiple compiler versions rather than a specific one.

Recommendation:

We advise that the compiler version is instead locked at the lowest version possible that the contract can be compiled at. For example, for version `v0.6.8` the contract should contain the following line:

```
pragma solidity 0.6.8;
```

Alleviation:

The development team opted to consider our references and locked the compiler to version `0.6.12`.

IPL-01: Unlocked Compiler Version

Type	Severity	Location
Language Specific	Informational	IPool.sol L3

Description:

The contract has unlocked compiler version. An unlocked compiler version in the source code of the contract permits the user to compile it at or above a particular version. This, in turn, leads to differences in the generated bytecode between compilations due to differing compiler version numbers. This can lead to an ambiguity when debugging as compiler specific bugs may occur in the codebase that would be hard to identify over a span of multiple compiler versions rather than a specific one.

Recommendation:

We advise that the compiler version is instead locked at the lowest version possible that the contract can be compiled at. For example, for version `v0.6.8` the contract should contain the following line:

```
pragma solidity 0.6.8;
```

Alleviation:

The development team opted to consider our references and locked the compiler to version `0.6.12`.

Appendix

Finding Categories

Gas Optimization

Gas Optimization findings refer to exhibits that do not affect the functionality of the code but generate different, more optimal EVM opcodes resulting in a reduction on the total gas cost of a transaction.

Mathematical Operations

Mathematical Operation exhibits entail findings that relate to mishandling of math formulas, such as overflows, incorrect operations etc.

Logical Issue

Logical Issue findings are exhibits that detail a fault in the logic of the linked code, such as an incorrect notion on how `block.timestamp` works.

Control Flow

Control Flow findings concern the access control imposed on functions, such as owner-only functions being invoke-able by anyone under certain circumstances.

Volatile Code

Volatile Code findings refer to segments of code that behave unexpectedly on certain edge cases that may result in a vulnerability.

Data Flow

Data Flow findings describe faults in the way data is handled at rest and in memory, such as the result of a `struct` assignment operation affecting an in-memory `struct` rather than an in-storage one.

Language Specific

Language Specific findings are issues that would only arise within Solidity, i.e. incorrect usage of `private` or `delete`.

Coding Style

Coding Style findings usually do not affect the generated byte-code and comment on how to make the codebase more legible and as a result easily maintainable.

Inconsistency

Inconsistency findings refer to functions that should seemingly behave similarly yet contain different code, such as a `constructor` assignment imposing different `require` statements on the input variables than a setter function.

Magic Numbers

Magic Number findings refer to numeric literals that are expressed in the codebase in their raw format and should otherwise be specified as `constant` contract variables aiding in their legibility and maintainability.

Compiler Error

Compiler Error findings refer to an error in the structure of the code that renders it impossible to compile using the specified version of the project.

Dead Code

Code that otherwise does not affect the functionality of the codebase and can be safely omitted.