Security Assessment

Zild Lend

May 1st, 2021

Summary

This report has been prepared for Zild Lend smart contracts, to discover issues and vulnerabilities in the source code of their Smart Contract as well as any contract dependencies that were not part of an officially recognized library. A comprehensive examination has been performed, utilizing Dynamic Analysis, Static Analysis, and Manual Review techniques.

The auditing process pays special attention to the following considerations:

- Testing the smart contracts against both common and uncommon attack vectors.
- Assessing the codebase to ensure compliance with current best practices and industry standards.
- Ensuring contract logic meets the specifications and intentions of the client.
- Cross referencing contract structure and implementation against similar smart contracts produced by industry leaders.
- Thorough line-by-line manual review of the entire codebase by industry experts.

The security assessment resulted in findings that ranged from critical to informational. We recommend addressing these findings to ensure a high level of security standards and industry practices. We suggest recommendations that could better serve the project from the security perspective:

- Enhance general coding practices for better structures of source codes;
- Add enough unit tests to cover the possible use cases given they are currently missing in the repository;
- Provide more comments per each function for readability, especially contracts are verified in public;
- Provide more transparency on privileged activities once the protocol is live.

Overview

Project Summary

Project Name	Zild Lend
Description	Zild is a blockchain-based money market protocol.
Platform	Ethereum
Language	Solidity
Codebase	https://github.com/zildfinance/zild_lend
Commits	d75df190560545d8a0f3e3b642713e7289fb9339

Audit Summary

Delivery Date	May 01, 2021
Audit Methodology	Static Analysis, Manual Review
Key Components	

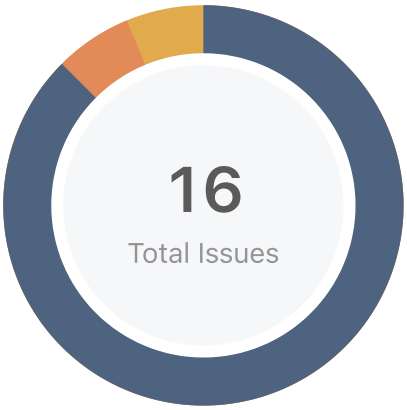
Vulnerability Summary

Total Issues	16
Critical	0
Major	1
Minor	1
Informational	14
Discussion	0

Audit Scope

ID	file	SHA256 Checksum
CME	CarefulMath.sol	289cb4e08e8f69193ba633a14b2bff5b4859d9899506b5441b0f0fd86607280d
COM	Comptroller.sol	201fd50dea8dd68a09bccebac08d6d639f5c1b7229d785ade6452199a250135a
CIE	ComptrollerInterface.sol	887408c6b8727b845b9a9d221ad861f7637ee3e64f2e782cbf28ab6532c35273
CSE	ComptrollerStorage.sol	c64bb1b7e1df994c0b82b6aa26eda66f152f97938d138a6c649d4f8b77eb6cec
EIP	EIP20Interface.sol	5a1a793d4171877c515deaa6121015bb106935def3026b63ca3cd440266e2fa3
EIN	EIP20NonStandardInterface.sol	28a059657b068902951e7318751cf631020367b250eb0db2c8c158b7498f903f
ERE	ErrorReporter.sol	c0b5f9832a0dbf7c6afabf3c7123d9eb31962d8929ce9f430061fa1b4d6911aa
EXP	Exponential.sol	2a5ba7aad28f3353b978d1768355bc84ce379c7e5e37d5f6f4e23b6949f47ee3
GTE	GToken.sol	b9d1e0945b1b146adde91375c9239fa3b075f5c0b1bebd15a2811fe725de862d
GTI	GTokenInterfaces.sol	974c521a48ab93c3e7d704ea615e8120011ddccab7517854b821e2c9b3441435
IRM	InterestRateModel.sol	2dea6a5512cc7e19f98db018dbd036548eee60e78ec99bfa0c4b5268b7e7bb1d
POE	PriceOracle.sol	2ebdf3f607319f54aa51d2dbe707421f6c4219992239f42b4f01e496e4f65150
SME	SafeMath.sol	caacc05116f311cf89a3dc196fe135486d09bc62c9eff61a68032f45dbb2a8fe
UPO	UniswapPriceOracleV2.sol	a2c190ab51df37ca2779b68c08a1a751b705a86f31a575e43b0bdb7b48e02127
WPI	WhitePaperInterestRateModel.sol	93d92b65d82c340e452d6f17cb6d5c5df9ec6fed42a91a433ac98fa19baab3d5
ZET	ZETH.sol	6844d983f7c102d6f37443506c66b603afb1d99c7dcf72b3dc8314b1dce5aca5
ZEE	ZErc20.sol	a9b620854142699652f237d33f6ad68841452fe2541181c5124069810a44d958

Findings



Critical	0 (0.00%)
Major	1 (6.25%)
Minor	1 (6.25%)
Informational	14 (87.50%)
Discussion	0 (0.00%)

ID	Title	Category	Severity	Status
COM-01	Boolean Equality	Gas Optimization	Informational	Acknowledged
COM-02	Misuse of a Boolean Constant	Coding Style	Informational	Acknowledged
COM-03	Return value not stored	Gas Optimization	Informational	Acknowledged
COM-04	Unlocked Compiler Version Declaration	Language Specific	Informational	Resolved
COM-05	Proper Usage of “public” and “external” type	Coding Style	Informational	Acknowledged
COM-06	Incorrect Naming Convention Utilization	Coding Style	Informational	Partially Resolved
COM-07	Unused Variable	Gas Optimization	Informational	Acknowledged
COM-08	Incorrect Naming Convention Utilization	Logical Issue	Informational	Acknowledged
GTE-01	Proper Usage of “public” and “external” type	Coding Style	Informational	Acknowledged
GTE-02	Incorrect Naming Convention Utilization	Coding Style	Informational	Partially Resolved
GTE-03	Missing Some Important Checks	Logical Issue	Minor	Partially Resolved
GTE-04	Redundant Codes	Coding Style	Informational	Acknowledged

ID	Title	Category	Severity	Status
GTE-05	Logical Issues of Functions `_setBorrowFee()` and `_setRedeemFee()`	**Centralization / Privilege**	**Major**	**Resolved**
GTE-06	Incorrect Naming Convention Utilization	Logical Issue	Informational	Acknowledged
UPO-01	Missing checks of array length	Gas Optimization	Informational	Acknowledged
UPO-02	Incorrect Formula	Logical Issue	Informational	Acknowledged

COM-01 | Boolean Equality

Category	Severity	Location	Status
Gas Optimization	Informational	Comptroller.sol: 183, 1073, 1083, 1092, 1101	Acknowledged

Description

Boolean constants can be used directly and do not need to be compared to true or false.

Example:

```
183   if (marketToJoin.accountMembership[borrower] == true) {
184               // already joined
185               return Error.NO_ERROR;
186           }
```

Recommendation

Consider changing it as following:

```
  if (marketToJoin.accountMembership[borrower]) {...
…}
```

COM-02 | Misuse of a Boolean Constant

Category	Severity	Location	Status
Coding Style	Informational	Comptroller.sol: 305, 430, 483, 555, 616, 659	Acknowledged

Description

Boolean constants in code have only a few legitimate uses. Other uses (in complex expressions, as conditionals) indicate either an error or, most likely, the persistence of faulty code.

Examples:

```
305 if (false) {
306             maxAssets = maxAssets;
307 }
```

Recommendation

Consider removing the ineffectual code.

COM-03 | Return value not stored

Category	Severity	Location	Status
Gas Optimization	Informational	Comptroller.sol: 1030	Acknowledged

Description

The return value of an external call is not stored in a local or state variable.

Examples:

```
function _supportMarket(CToken cToken) external returns (uint) {
...
gToken.isCToken();
...
}
```

Recommendation

Ensure that all the return values of the function calls are used.

Consider adding “require” statement for isCToken:

```
require(gToken.isCToken(),"This is not a GToken contract!");
```

COM-04 | Unlocked Compiler Version Declaration

Category	Severity	Location	Status
Language Specific	Informational	Comptroller.sol: 1	Resolved

Description

The compiler version utilized throughout the project uses the "^" prefix specifier, denoting that a compiler version which is greater than the version will be used to compile the contracts.

Recommendation

It is a general practice to instead lock the compiler at a specific version rather than allow a range of compiler versions to be utilized to avoid compiler-specific bugs and be able to identify ones more easily. We recommend locking the compiler at the lowest possible version that supports all the capabilities wished by the codebase. This will ensure that the project utilizes a compiler version that has been in use for the longest time and as such is less likely to contain yet-undiscovered bugs.

Alleviation

The team heeded our advice and resolved this issue in commit bc14c5ba400bdcd249534a01e76329837fccbabd.

COM-05 | Proper Usage of “public” and “external” type

Category	Severity	Location	Status
Coding Style	Informational	Comptroller.sol: 83, 106, 156, 690, 716, 870, 1053, 1070, 1080, 1090, 1099, 1127, 1131	Acknowledged

Description

“public” functions that are never called by the contract should be declared “external” . When the inputs are arrays, “external” functions are more efficient than “public” functions.

Examples:

Functions like : `_setPendingAdmin()`, `_acceptAdmin()`, `enterMarkets()`, `getAccountLiquidity()`, `getHypotheticalAccountLiquidity()`, `_setPriceOracle()`, `_setPauseGuardian()`, `_setMintPaused()`, `_setBorrowPaused()`, `_setTransferPaused()`, `_setSeizePaused()`, `getAllMarkets()`, `getBlockNumber()`, `initialize()`, `_setInterestRateModel()`

Recommendation

Consider using the “external” attribute for functions never called from the contract.

COM-06 | Incorrect Naming Convention Utilization

Category	Severity	Location	Status
Coding Style	Informational	Comptroller.sol: 59, 62, 65, 68, 71	Partially Resolved

Description

Solidity defines a naming convention that should be followed. In general, the following naming conventions should be utilized in a Solidity file:

Constants should be named with all capital letters with underscores separating words UPPER_CASE_WITH_UNDERSCORES

Refer to https://solidity.readthedocs.io/en/v0.5.17/style-guide.html#naming-conventions

Examples:

Constants like: `projectHash`, `tenThousand`, `closeFactorMinMantissa`, `closeFactorMaxMantissa`, `collateralFactorMaxMantissa`, `liquidationIncentiveMinMantissa`, `liquidationIncentiveMaxMantissa`

Recommendation

The team heeded our advice and partially resolved this issue in commit bc14c5ba400bdcd249534a01e76329837fccbabd.

COM-07 | Unused Variable

Category	Severity	Location	Status
Gas Optimization	Informational	Comptroller.sol: 571, 572, 629	Acknowledged

Description

Some unused function parameters are declared. Remove or comment out the variable name.

Examples: Function parameters “liquidator”, “borrower” and “dst”

Recommendation

Consider removing the unused function parameters.

COM-08 | Incorrect Naming Convention Utilization

Category	Severity	Location	Status
Logical Issue	Informational	Comptroller.sol: 1030	Acknowledged

Description

Solidity defines a naming convention that should be followed.

Refer to: https://solidity.readthedocs.io/en/v0.7.1/style-guide.html#naming-conventions

Check the below example in contract `GToken`, there are many cToken instead of gToken. Better to keep the coding style consistent.

Examples:

```
302 uint cTokenBalance = accountTokens[account];
```

```
1143 if (address(cTokenCollateral) == address(this)) {
1144     seizeError = seizeInternal(address(this), liquidator, borrower, seizeTokens);
1145 } else {
1146     seizeError = cTokenCollateral.seize(liquidator, borrower, seizeTokens);
1147 }
```

```
1030 gToken.isCToken(); // Sanity check to make sure its really a GToken
```

Recommendation

The recommendations outlined here are intended to improve the readability, and thus they are not rules, but rather guidelines to try and help convey the most information through the names of things.

GTE-01 | Proper Usage of “public” and “external” type

Category	Severity	Location	Status
Coding Style	Informational	GToken.sol: 60, 1487	Acknowledged

Description

“public” functions that are never called by the contract should be declared “external” . When the inputs are arrays, “external” functions are more efficient than “public” functions.

Examples:

Functions like : `_setPendingAdmin()`, `_acceptAdmin()`, `enterMarkets()`, `getAccountLiquidity()`, `getHypotheticalAccountLiquidity()`, `_setPriceOracle()`, `_setPauseGuardian()`, `_setMintPaused()`, `_setBorrowPaused()`, `_setTransferPaused()`, `_setSeizePaused()`, `getAllMarkets()`, `getBlockNumber()`, `initialize()`, `_setInterestRateModel()`

Recommendation

Consider using the “external” attribute for functions never called from the contract.

GTE-02 | Incorrect Naming Convention Utilization

Category	Severity	Location	Status
Coding Style	Informational	GToken.sol: 17, 35	Partially Resolved

Description

Solidity defines a naming convention that should be followed. In general, the following naming conventions should be utilized in a Solidity file:

Constants should be named with all capital letters with underscores separating words UPPER_CASE_WITH_UNDERSCORES

Refer to https://solidity.readthedocs.io/en/v0.5.17/style-guide.html#naming-conventions

Examples:

Constants like: `projectHash`, `tenThousand`, `closeFactorMinMantissa`, `closeFactorMaxMantissa`, `collateralFactorMaxMantissa`, `liquidationIncentiveMinMantissa`, `liquidationIncentiveMaxMantissa`

Recommendation

The team heeded our advice and partially resolved this issue in commit bc14c5ba400bdcd249534a01e76329837fccbabd.

GTE-03 | Missing Some Important Checks

Category	Severity	Location	Status
Logical Issue	Minor	GToken.sol: 91, 144	Partially Resolved

Description

Functions `initialize()`, `_setFeeTo()` on the afore-mentioned lines are missing parameter validations.

Recommendation

Consider adding checks as following example:

```
require(feeTo_ != address(0), "feeTo_ is zero address");
```

Alleviation

The team heeded our advice and partially resolved this issue in commit bc14c5ba400bdcd249534a01e76329837fccbabd.

GTE-04 | Redundant Codes

Category	Severity	Location	Status
Coding Style	Informational	GToken.sol: 17	Acknowledged

Description

Variable `projectHash` is defined but never used.

Recommendation

Consider removing the redundant codes.

GTE-05 | Logical Issues of Functions `_setBorrowFee()` and `_setRedeemFee()`

Category	Severity	Location	Status
Centralization / Privilege	Major	GToken.sol: 105, 125	Resolved

Description

Functions `_setBorrowFee()` and `_setRedeemFee()` can only be called by admin. Better to set a range for fee rate. Otherwise the rate can be changed anytime with unreasonable fee rate.

Recommendation

Consider to set a range for the fee rate or move the abovementioned functions to the execution queue of the Timelock contract.

Alleviation

The team heeded our advice and resolved this issue in commit bc14c5ba400bdcd249534a01e76329837fccbabd.

GTE-06 | Incorrect Naming Convention Utilization

Category	Severity	Location	Status
Logical Issue	Informational	GToken.sol: 302, 1143	Acknowledged

Description

Solidity defines a naming convention that should be followed.

Refer to: https://solidity.readthedocs.io/en/v0.7.1/style-guide.html#naming-conventions

Check the below example in contract `GToken`, there are many cToken instead of gToken. Better to keep the coding style consistent.

Examples:

```
302 uint cTokenBalance = accountTokens[account];
```

```
1143 if (address(cTokenCollateral) == address(this)) {
1144     seizeError = seizeInternal(address(this), liquidator, borrower, seizeTokens);
1145 } else {
1146     seizeError = cTokenCollateral.seize(liquidator, borrower, seizeTokens);
1147 }
```

```
1030 gToken.isCToken(); // Sanity check to make sure its really a GToken
```

Recommendation

The recommendations outlined here are intended to improve the readability, and thus they are not rules, but rather guidelines to try and help convey the most information through the names of things.

UPO-01 | Missing checks of array length

Category	Severity	Location	Status
Gas Optimization	Informational	UniswapPriceOracleV2.sol: 209, 689	Acknowledged

Description

Many arrays such as "gTokens_", "underlyings_" and "symbolHashs_" are used to initialize the “TokenConfig” array. It’s better to ensure the length of all these arrays are exactly the same.

Recommendation

Consider adding some checks as following:

```
require(gTokens_.length == underlyings_.length, "array length not same");
require(underlyings_.length == symbolHashs_.length, "array length not same");
…
```

UPO-02 | Incorrect Formula

Category	Severity	Location	Status
Logical Issue	Informational	UniswapPriceOracleV2.sol: 852	Acknowledged

Description

According to the formula in the function `fetchAnchorPriceETH()`, the formula in the function `fetchAnchorPriceUSD()` is incorrect.

```
852  uint unscaledPriceMantissa = mul(rawUniswapPriceMantissa, config.baseUnit);
853  //uint anchorPrice = mul(unscaledPriceMantissa , 1e6) / usdtBaseUnit / expScale;
854  uint anchorPrice = unscaledPriceMantissa  / expScale;
```

Recommendation

Consider changing it as following example:

```
uint unscaledPriceMantissa = mul(rawUniswapPriceMantissa, 1e6);
//uint anchorPrice = mul(unscaledPriceMantissa , config.baseUnit) / usdtBaseUnit /
expScale;
uint anchorPrice = mul(rawUniswapPriceMantissa , config.baseUnit)  / expScale;
```

Alleviation

The recommendation was not taken into account, with the ZILD team stating "We did some tests, the code is correct.".

Appendix

Finding Categories

Gas Optimization

Gas Optimization findings 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 findings relate to mishandling of math formulas, such as overflows, incorrect operations etc.

Logical Issue

Logical Issue findings 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 but rather 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.

Disclaimer

This report is subject to the terms and conditions (including without limitation, description of services, confidentiality, disclaimer and limitation of liability) set forth in the Services Agreement, or the scope of services, and terms and conditions provided to the Company in connection with the Agreement. This report provided in connection with the Services set forth in the Agreement shall be used by the Company only to the extent permitted under the terms and conditions set forth in the Agreement. This report may not be transmitted, disclosed, referred to or relied upon by any person for any purposes without CertiK's prior written consent.

This report is not, nor should be considered, an "endorsement" or "disapproval" of any particular project or team. This report is 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 assessment. This report does 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.

This report should not be used in any way to make decisions around investment or involvement with any particular project. This report in no way provides investment advice, nor should be leveraged as investment advice of any sort. This report represents an extensive assessing 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.

About

Founded in 2017 by leading academics in the field of Computer Science from both Yale and Columbia University, CertiK is a leading blockchain security company that serves to verify the security and correctness of smart contracts and blockchain-based protocols. Through the utilization of our world-class technical expertise, alongside our proprietary, innovative tech, we're able to support the success of our clients with best-in-class security, all whilst realizing our overarching vision; provable trust for all throughout all facets of blockchain.