SOLIDIFIED

Audit Report for TrueFi - February 7, 2022

Summary

Audit Report prepared by Solidified covering the Truefi Ethereum smart contracts.

Process and Delivery

Three (3) independent Solidified experts performed an unbiased and isolated audit of the code. The debrief on 7 February 2022.

Audited Files

The source code has been supplied in the form of two GitHub repositories:

https://github.com/trusttoken/truefi/

Commit number: `efad960c9b0069758e44b662979f87e18ca4810a`

The scope of the audit was limited to the following files:

```
packages/contracts/contracts/ragnarok
├── BorrowerSignatureVerifier.sol
├── BulletLoans.sol
├── ManagedPortfolio.sol
├── ManagedPortfolioFactory.sol
├── ProtocolConfig.sol
├── SignatureOnlyLenderVerifier.sol
├── access
│   ├── InitializableManageable.sol
│   └── Manageable.sol
└── proxy
    └── ProxyWrapper.sol
```

Intended Behavior

The smart contracts implement an uncollateralized lending protocol that allows asset managers to create lending pools in which liquidity providers deposit funds. Asset managers can also issue loans to borrowers, which should be repaid by a certain date.

Code Complexity and Test Coverage

Smart contract audits are an important step to improve the security of smart contracts and can find many issues. However, auditing complex codebases has its limits and a remaining risk is present (see disclaimer).

Users of a smart contract system should exercise caution. In order to help with the evaluation of the remaining risk, we provide a measure of the following key indicators: **code complexity**, **code readability**, **level of documentation**, and **test coverage**.

Note, that high complexity or lower test coverage does equate to a higher risk. Certain bugs are more easily detected in unit testing than in a security audit and vice versa. It is, therefore, more likely that undetected issues remain if the test coverage is low or non-existent.

Criteria	Status	Comment
Code complexity	Medium	-
Code readability and clarity	High	-
Level of Documentation	High	-
Test Coverage	High	-

Issues Found

Solidified found that the TrueFi contracts contain no critical issue, 6 major issues, 9 minor issues in addition to 3 informational notes.

In addition, one end-user warning has been added.

We recommend all issues are amended, while the notes are up to the team's discretion, as they refer to best practices.

Issue #	Description	Severity	Status
1	Portfolio managers can access and remove all funds	Warning	-
2	ManagedPortfolio.sol: Deposits will eventually fail when too many loans have been created for a particular portfolio	Major	Pending
3	ManagedPortfolio.sol: ERC-20 return values are ignored	Major	Pending
4	BorrowerSignatureVerifier.sol: Signatures can be replayed multiple times	Major	Pending
5	ManagedPortfolio.sol: Lending pool shares are calculated without taking into account the newly deposited amount	Major	Pending
6	ManagedPortfolio.sol: A malicious contract can potentially bypass lender verification in function deposit()	Major	Pending
7	BulletLoans.sol: Contract BulletLoans is incorrectly initializing ERC721	Major	Pending
8	BulletLoans.sol: Managers can mark repaid loans as defaulted	Minor	Pending
9	BulletLoans.sol and ManagedPortfolio.sol: Loans can be marked as defaulted even before repay date	Minor	Pending

10	BulletLoans.sol and ManagedPortfolio.sol: Missing guards on zero duration and zero amount loans	Minor	Pending
11	BulletLoans.sol: Loan status is not updated if parameters are changed with overloaded function	Minor	Pending
12	BulletLoans.sol: Missing event emission when updating loan parameters	Minor	Pending
13	ManagedPortfolio.sol: Function withdraw() fails to update totalDeposited	Minor	Pending
14	BulletLoans.sol: Function initialize() does not validate _borrowerSignatureVerifier	Minor	Pending
15	BulletLoans.sol: Function updateLoanParameters() should only be allowed to update ‘Issued’ loans	Minor	Pending
16	BulletLoans.sol: Loan repayments can be greater than owned amount	Minor	Pending
17	BulletLoans.sol: Anyone can create a loan entry	Note	-
18	ManagedPortfolio.sol: Gas inefficiencies due to redundant checks	Note	-
19	Absence of zero address validation	Note	-

Warnings

1. Portfolio managers can access and remove all funds

The portfolio manager role is extremely powerful. A manager can do the following:

- Create loans to any address, including themselves and other addresses owned by themselves.
- Change the fees at will.
- Mark loans as defaulted at will.

This essentially means that the manager can steal funds at will.

Recommendation
Consider adding the following safeguards:

- Not allowing fees to be modified after portfolio creation or placing bounds on fees
- Add checks on loans being declared defaulted (see issue below)
- Implementing a whitelisting/KYC procedure for borrowers (not just lenders and managers)
- Not allowing managers to issue loans to themselves

Critical Issues

No critical Issues found.

Major Issues

2. ManagedPortfolio.sol: Deposits will eventually fail when too many loans have been created for a particular portfolio

Loans in a portfolio are managed in a variable sized array which grows in size. The `public view` function `illiquidValue()` iterates over the whole array to calculate the total outstanding value. This is fine in read-only calls.
However, the `deposit()` function uses the function in a state changing transaction. Since the array will get larger with each loan issued, this transaction will eventually hit the block gas limit and always revert.
This issue is made more problematic due to the fact that operations that loans are never removed from the data-structure.

Recommendation
Consider keeping track of the illiquid value in a separate variable to avoid iterating over the unbound array.

3. ManagedPortfolio.sol: ERC-20 return values are ignored

Throughout the contract `transfer()` and `transferFrom()` calls are made on the underlying token without checking the return value of the call. Most tokens revert on failure, but some implementations, including well-known tokens return `false` instead of reverting. This may lead to failed transfers being treated as successful.

Recommendation
Consider checking the return types and/or use OpenZeppelin's `safeERC20` implementation (https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20/utils/SafeERC20.sol)

4. `BorrowerSignatureVerifier.sol`: Signatures can be replayed multiple times

The function `verify()` verifies off-chain ECDA signatures for borrowers authorizing changes to existing loans. Whilst the signature verification is correct, the signatures can be replayed at will. This means that an old authorization on a particular loan may be used again at any time in the future on that loan. This is due to the signed message not including a uniqueness value, such as a nonce.

Recommendation

Consider keeping track of signature nonces per borrower address or loan id and adding these nonces to the message being signed.

5. `ManagedPortfolio.sol`: Lending pool shares are calculated without taking into account the newly deposited amount

The function `deposit()` function calculates and mints the newly emitted shares before transferring the deposited funds to the contract. This means that calculation performed by `getAmountToMint()` does not take into account the newly deposited amount, since it uses the contract's balance to calculate the liquid amount through calls to `value()` and `liquidValue()`. This leads to incorrect share emission.

Recommendation

Consider reversing the order of operations. **Note, that performing the token transfer first would open the protocol up to potential reentrancy issues with malicious token implementation. A reentrancy guard is recommended in this case.**

6. ManagedPortfolio.sol: A malicious contract can potentially bypass lender verification in function deposit()

In case `ManagedPortfolio.lenderVerifier` is an instance of `SignatureOnlyLenderVerifier` (which is the currently active strategy according to documentation), all the lender needs to do to bypass deposit verification is call `deposit()` from a malicious contract that implements the function `isValidSignature()` as follows:

```
function isValidSignature(bytes32, bytes memory) external pure returns (bytes4) {
    return bytes4(keccak256("isValidSignature(bytes32,bytes)"));
}
```

This is due to the fact that `SignatureOnlyLenderVerifier` uses the `SignatureValidator` library, which assumes that the given contract is an instance of `IVerifier` and in turn queries its `isValidSignature()` function for verification.

Recommendation

The `SignatureValidator` library should not assume that the given `signer` contract is an instance of `IVerifier`.

7. BulletLoans.sol: Contract BulletLoans is incorrectly initializing ERC721

The contract `BulletLoans` is incorrectly initializing contract `ERC721` via its constructor. Since `BulletLoans` is designed to be accessed via the *proxy pattern*, any initializations done via its constructor will be written to incorrect storage, and are thus completely irrelevant to the proxy contract.

Recommendation

Consider using OpenZeppelin's `ERC721Upgradeable` instead of `ERC721`, and call `ERC721Upgradeable.__ERC721_init()` from within `BulletLoans.initialize()` in order to correctly initialize the contract. Function `BulletLoans.initialize()` should also add additional parameters for `name_` and `symbol_`, which are required by `ERC721Upgradeable`.

Minor Issues

8. BulletLoans.sol: Managers can mark repaid loans as defaulted

The function `markeLoanAsDefaulted()` allows the manager to mark a loan as defaulted even if the loan is fully repaid. The function's require statement ensures the loan's status to not be defaulted but fails to check if the loan is completely repaid.

Recommendation

Consider allowing loans to be marked as defaulted only if they have not been repaid.

9. BulletLoans.sol and ManagedPortfolio.sol: Loans can be marked as defaulted even before repay date

The functions `markLoanAsDefaulted()` in both contracts allow a portfolio manager to mark a loan as defaulted. However, there are no checks to confirm that the loan's repayment date has expired. This means that a manager can mark a loan as defaulted at any time.

Recommendation

Consider allowing loans to be marked as defaulted only once their repay date has passed.

10. BulletLoans.sol and ManagedPortfolio.sol: Missing guards on zero duration and zero amount loans

The functions `creatBulletLoan()` and `createLoan()` allow the creation of loans of duration 0 or with an amount of 0.

Recommendation

Consider adding guards to avoid filling up the data structure with empty loans.

11. BulletLoans.sol: Loan status is not updated if parameters are changed with overloaded function

The function `updateLoanParameters()` has two implementations. The second implementation, which allows the borrower to authorize the update with an off-chain signature fails to update the loan status. This is likely to be the case because the function is only intended to be used to increase the debt or lower repayment date. However, this condition is not enforced.

Recommendation
Consider checking and updating the loan status or enforcing the condition mentioned above. If the former approach is chosen an event should be emitted (see issue below).

12. BulletLoans.sol: Missing event emission when updating loan parameters

The function `updateLoanParameters()` might lead to the loan status changing. In this case, the implementation should emit a `LoanStatusChanged` event. The lack of this event might lead to off-chain components missing the status change.

Recommendation
Consider emitting the event.

13. ManagedPortfolio.sol: Function withdraw() fails to update totalDeposited

`totalDeposited` should be updated whenever any underlying tokens are withdrawn, otherwise function `deposit()` could potentially prevent deposits due to `totalDeposited` being incorrectly greater than `maxSize`.

Recommendation
Add the statement: `totalDeposited -= amountToWithdraw` to function `withdraw()`.

14. BulletLoans.sol: Function initialize() does not validate _borrowerSignatureVerifier

The function `initialize()` does not validate that `_borrowerSignatureVerifier` is a valid contract address. This can potentially cause function `updateLoanParameters()` to always revert.

Recommendation

Require that `_borrowerSignatureVerifier` is a contract address.

15. BulletLoans.sol: Function updateLoanParameters() should only be allowed to update Issued loans

Recommendation

Require that `loan.status == BulletLoanStatus.Issued` before updating the loan parameters.

16. BulletLoans.sol: Loan repayments can be greater than owned amount

The function `repay()` allows borrowers to repay more than they owe. Whilst this is not a security issue, it may lead to accidental loss of funds due to user error.

Recommendation

Consider enforcing that borrowers can only repay up to the owed amount.

Informational Notes

17. BulletLoans.sol: Anyone can create a loan entry

The function `createLoan()` is unprotected and can be called by anyone. This does not result in any funds being allocated. However, it results in the minting of loan NFTs and the emission of an event that could confuse external systems that use this information to monitor the protocol. In addition, allowing minting of loan NFTs from non-whitelisted sources makes it easier for fraudulent copycat portfolios to appear more legitimate.

Recommendation

Consider placing a guard on the function to only allow loan minting from authorized portfolios

18. ManagedPortfolio.sol: Gas inefficiencies due to redundant checks

The functions `deposit()` and `createBulletLoan()` check for the portfolio end time not having passed. However, in both cases, this is already enforced by the status checks on open portfolios (through the `onlyOpened` modifier and an explicit check respectively).

Recommendation

Consider removing unnecessary checks to save gas.

19. Absence of Zero Address validation

The contracts `BulletLoan.sol` and `ProtocolConfig.sol` include functions, i.e., `createLoans()` & `setProtocolAddress()` respectively, that update the state of crucial addresses in the contract but do not include any zero address validations

Recommendation

Consider adding `require()` statements to ensure only valid addresses are passed as arguments.

Disclaimer

Solidified audit is not a security warranty, investment advice, or an endorsement of TrustToken or its products. This audit does not provide a security or correctness guarantee of the audited smart contract. Securing smart contracts is a multistep process, therefore running a bug bounty program as a complement to this audit is strongly recommended.

The individual audit reports are anonymized and combined during a debrief process, in order to provide an unbiased delivery and protect the auditors from legal and financial liability.

Oak Security GmbH