CERTIK

Audit Report

Produced by CertiK

for

10th Feb, 2020

CertiK Audit Report For bZx

Request Date: 2020-02-05
Revision Date: 2020-02-10

Contents

Disclaimer

About CertiK

CertiK is a technology-led blockchain security company founded by Computer Science professors from Yale University and Columbia University built to prove the security and correctness of smart contracts and blockchain protocols.

CertiK, in partnership with grants from IBM and the Ethereum Foundation, has developed a proprietary Formal Verification technology to apply rigorous and complete mathematical reasoning against code. This process ensures algorithms, protocols, and business functionalities are secured and working as intended across all platforms.

CertiK differs from traditional testing approaches by employing Formal Verification to mathematically prove blockchain ecosystem and smart contracts are hacker-resistant and bug-free. CertiK uses this industry-leading technology together with standardized test suites, static analysis, and expert manual review to create a full-stack solution for our partners across the blockchain world to secure 6.2B in assets.

For more information: https://certik.org/

Manual Review Notes

Audit Scope

- Commit 36e42da78665d8da778c177d7d3a888d64230bb3, 2nd version:
  - LoanTokenLogicV4.sol#L192: Function `flashBorrowToken`
  - LoanTokenLogicV4.sol#L1402: Function `_checkPause`
  - .ArbitraryCaller.sol
- Commit 732548129be4e7ae8d8d576893053b256bc1223e, 1st version:
  - LoanTokenLogicV4.sol#192: Function `flashBorrowToken`
  - LoanTokenLogicV4.sol#1403: Function `_checkPause`
  - ArbitraryCaller@0x4c67b3dB1d4474c0EBb2DB8BeC4e345526d9E2fd

Provided Documentation

- Offcial Documentation: https://docs.bzx.network/
- GitHub Repository: https://github.com/bZxNetwork/bZx-monorepo

Results

CertiK has audited the bZx procotol's LoantokenlogicV4 for the flashBorrowToken feature. CertiK was unable to find any significant vulnerabilites in the aforementoned scope above, however we did not perform a comprehsive audit on the entire bZx protocol so we are only able to verify that what we reviewed does not contain any critical findings that could lead to immediately vulnerabilities. The review did raise a series of discussion points that we addressed with bZx. These can be found within the Potential Issues section at the bottom of our document.

Overview

Tokenized loans, called iTokens, are for letting the user lend their assets and earn interest by depositing into a global lending pool. The interest earned is proportional to the amount of iToken held by each lender. iTokens are minted by transferring the underlying asset to the contract, calling mint, and receiving back an equivalent amount of the iToken (ERC20) at the current iToken price. iTokens have an on-chain API to query current "redemption value" (`tokenPrice`), as well the interest rates paid by borrowers and paid to lenders.

Inheritance Analysis

The inheritance structurue of the `LoanTokenLogicV4` contract:

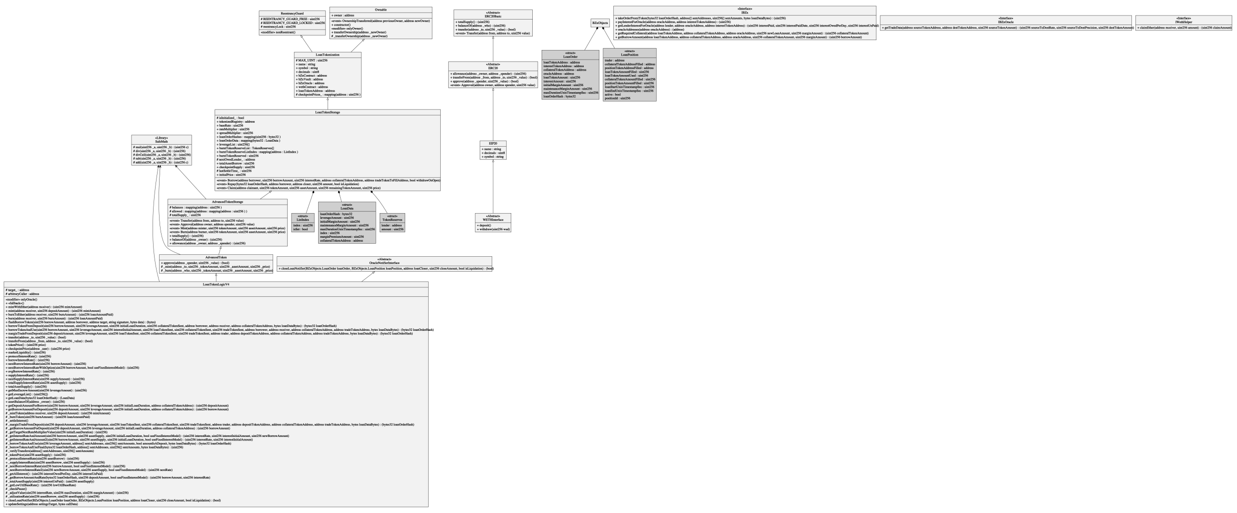


Figure 1: Inheritance Diagram for LoanTokenLogicV4

Call Analysis

The general call graph of the function `flashBorrowToken`:

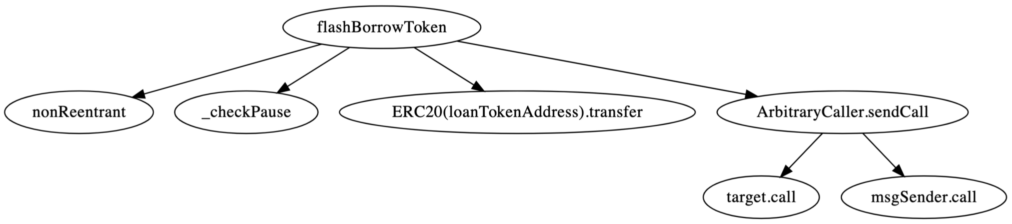


Figure 2: Call Graph of flashBorrowToken

The following sequential diagram indicates the core functionality of the `flashBorrowToken` function. A sucessful `flashBorrowToken` call:

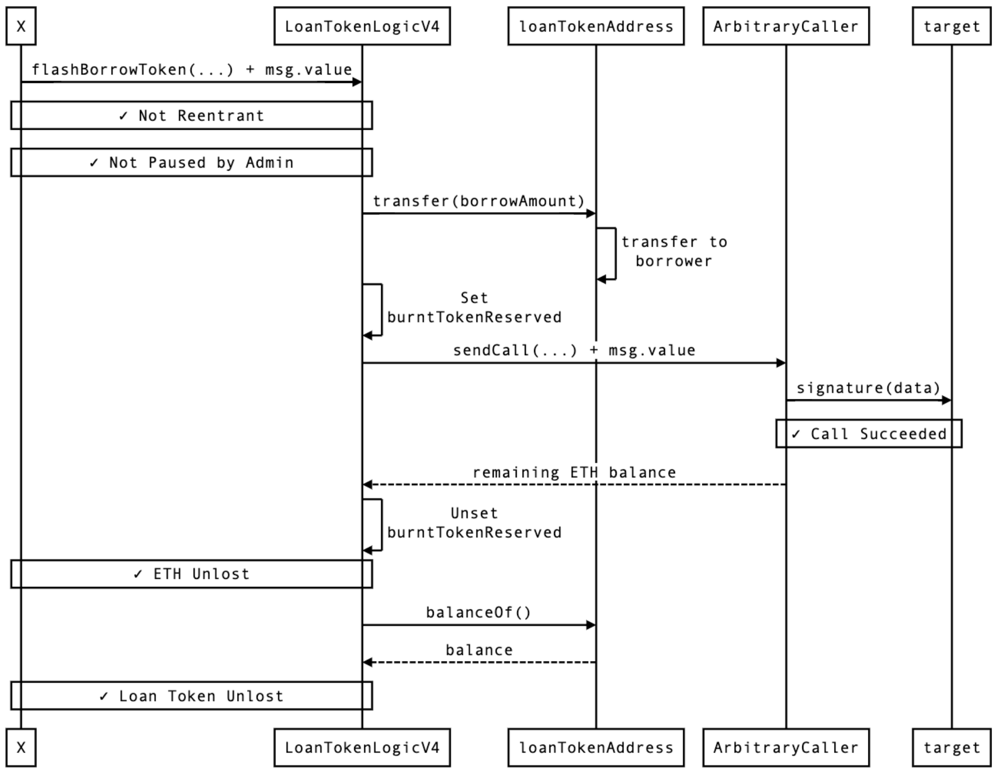


Figure 3: Sequence Diagram for flashBorrowToken

For reentrancy check that the `flashBorrowToken` function is protected by the OpenZeppelin `ReentrancyGuard`. The subsequent operations follow the check-effect-interactions pattern in general.

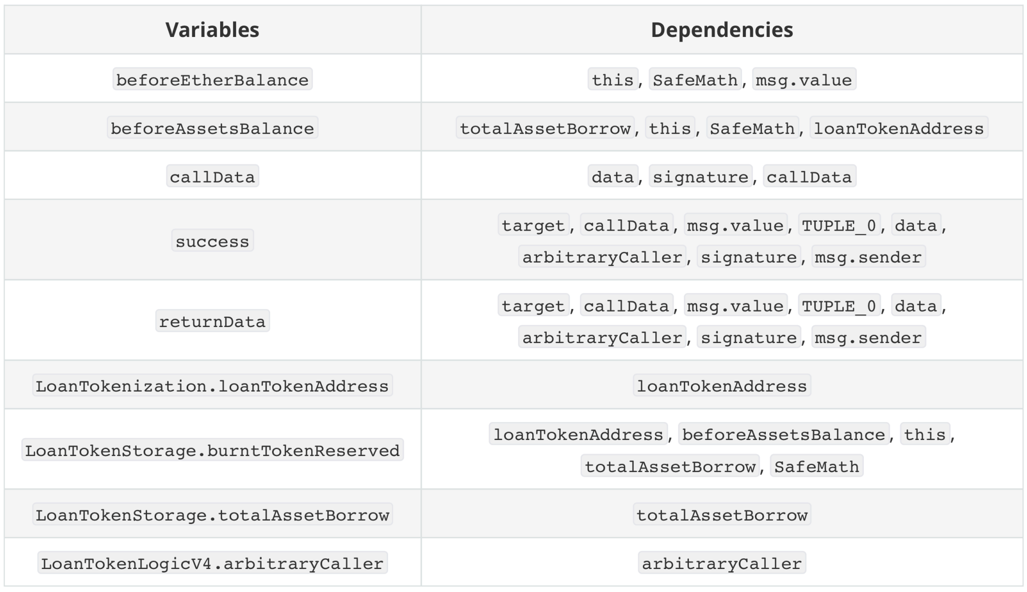
The pausing check corresponds to the `toggleFunctionPause` in the `LoanTokenSettingsLowerAdmin`. The `toggleFunctionPause` enables the admin to pause/unpause any other function providing the signature.

Here `X` denotes an unknown entity, and `target` is the argument supplied by `X`.

The argument `target` could be any entities (`X`, `LoanTokenLogicV4`, `loanTokenAddress`, `ArbitraryCaller`, or other ones). Possible situation for `target` - `target` being an account address, or contract that doesn't interact with bZx system: Account receives the ETH and produces internal side effects. - `target` being `ArbitraryCaller`: Prohibited by gas limit. - **Otherwise, the call from `ArbitraryCaller` may direct back to the `LoanTokenLogicV4`. Currently there are 34 functions that do not bear a `nonReentrant` guard and may be exploited by potential attackers.** See findings below.

Data Dependencies

Data dependency of `flashBorrowToken`:



Variables	Dependencies
`beforeEtherBalance`	`this`, `SafeMath`, `msg.value`
`beforeAssetsBalance`	`totalAssetBorrow`, `this`, `SafeMath`, `loanTokenAddress`
`callData`	`data`, `signature`, `callData`
`success`	`target`, `callData`, `msg.value`, `TUPLE_0`, `data`, `arbitraryCaller`, `signature`, `msg.sender`
`returnData`	`target`, `callData`, `msg.value`, `TUPLE_0`, `data`, `arbitraryCaller`, `signature`, `msg.sender`
`LoanTokenization.loanTokenAddress`	`loanTokenAddress`
`LoanTokenStorage.burntTokenReserved`	`loanTokenAddress`, `beforeAssetsBalance`, `this`, `totalAssetBorrow`, `SafeMath`
`LoanTokenStorage.totalAssetBorrow`	`totalAssetBorrow`
`LoanTokenLogicV4.arbitraryCaller`	`arbitraryCaller`

Figure 4: Data Dependency of flashBorrowToken

The `burntTokenReserved` is the only global state variable directly modified (no through subsequent external calls) by the `flashBorrowToken`.

Global State Variables

Global state variables related to the `flashBorrowToken` function:

- `loanTokenAddress` in LoanTokenization.sol: Address of the Loan Token.
- `burntTokenReserved` in LoanTokenStorage.sol:
  - An intermediate store of all potentially burnable tokens.
  - Conceptually the amount of lent out tokens + amount of lendable tokens.

Functional Dependencies

For completeness we provided two versions of a subgraph of all the functions that depend on `_totalAssetSupply()` (the sole direct dependent of the modified state variable `burntTokenReserved`).

The diagrams indicate the scope of the contract **that may be influenced by** the change of the `burntTokenReserved` in `flashBorrowToken`.

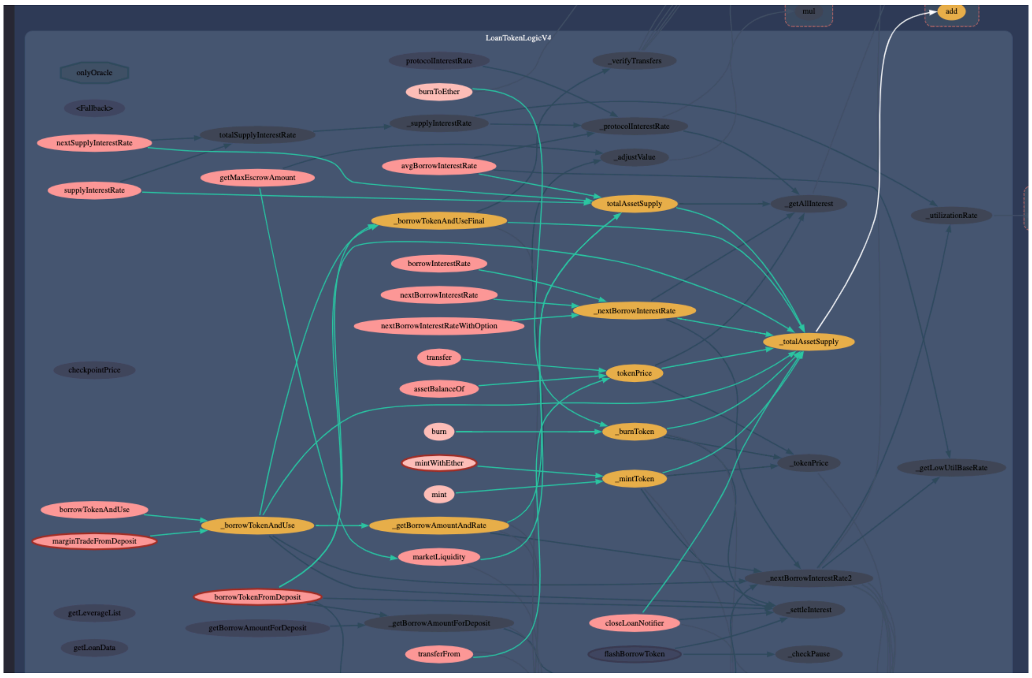


Figure 5: Call Graph for _totalAssetSupply

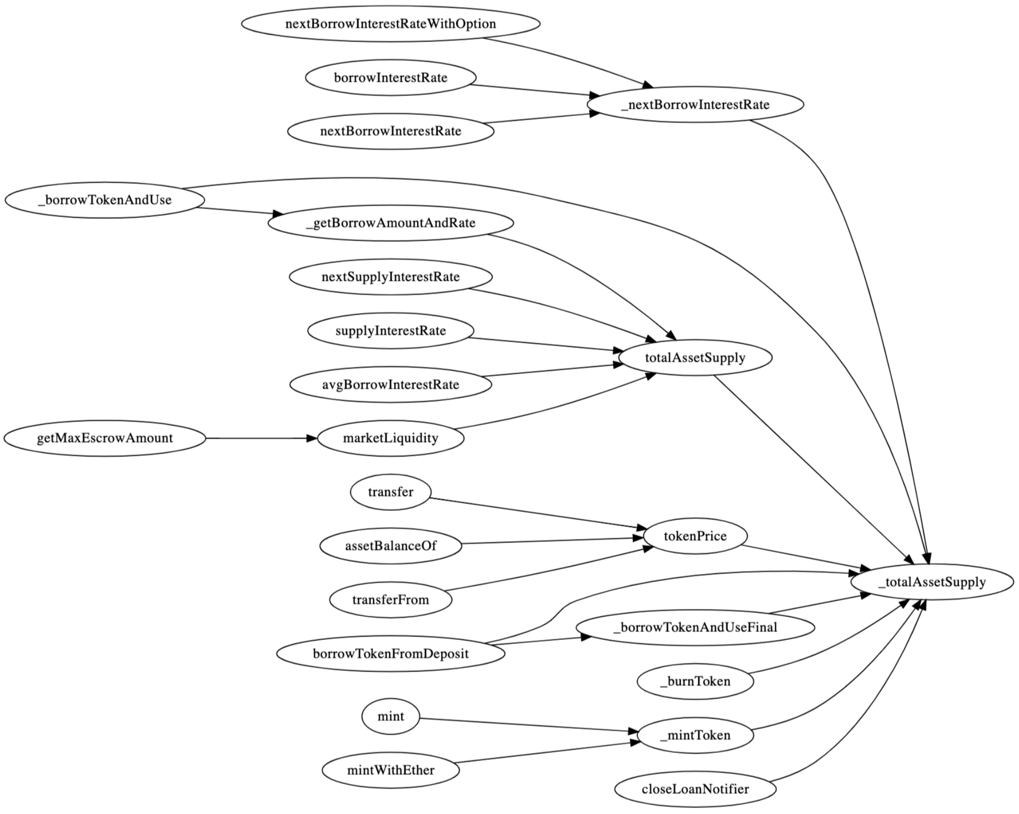


Figure 6: Simplified Call Graph for _totalAssetSupply

Potential Issues

- $\boxed{\text{INFO}}$ Since the arbirary caller `X` provides the target and call context for the arbitrary call, it can send all ETH back to itself without paying any fee, while receiving the Loan Token.
- $\boxed{\text{INFO}}$ `totalAssetBorrow` is not getting properly incremented in `flashBorrowToken` of `LoanTokenLogicV4` and doesn't reflect the borrow status of the contract. What would be the intended behavior of `totalAssetBorrow`?
- $\boxed{\text{INFO}}$ `burntTokenReserved` is set before and after the arbitrary external call, whereas due to the full coverage of `ReentrancyGuard` on every public/external function of the contract, all dependents of `burntTokenReserved` (as indicated in the functional dependencies subgraph above) may be affected. What would be the intended behavior of `burntTokenReserved`, and when should `_totalAssetSupply` be properly updated?
- $\boxed{\text{INFO}}$ The call from `ArbitraryCaller` may direct back to the `LoanTokenLogicV4`. Currently there are 34 functions that do not bear a `nonReentrant` guard and may be exploited by potential attackers.
- $\boxed{\text{INFO}}$ There have been situations where over and underflow can occur even despite using `SafeMath`, so it is recommended to compute upper and lower bounds based on the balance of the loan pool, and use those during up front parameter validation in all borrow functions.

Fixed in commit *36e42da78665d8da778c177d7d3a888d64230bb3*

- $\boxed{\text{INFO}}$ `ArbitraryCaller`: The transfer of remaining ETH at the end of the `sendCall` function seems to be redundant. The contract could end up with a positive balance after the call if it is deployed with initializing ETH. What would be the intended behavior of the `(success,)= msgSender.call.value(ethBalance)("")` call?

```
uint256 ethBalance = address(this).balance;
if (ethBalance != 0) {
    (success,) = msgSender.call.value(ethBalance)("");
}
require(success, ``eth refund failed");
```

  Analysis of `ArbitraryCaller`:

  1. Base: Assume the contract is deployed as-is, without any balance.
  2. If first four bytes of `data` contain anything other than `sendCall` sig, this will result in an `EVMC_INVALID`, reverting all state and consuming all gas.
  3. Else there are 4 ways in which `sendCall` can be called:
     – With a transaction.
     – With the `CALL` opcode - note that we are not assuming its the first message call, this will be important later on.
     – With the `CALLCODE` opcode or with the `DELEGATECALL` opcode.
       * will indeed make `ethBalance` (after the call; line 27) possibly > 0.
       * but this would apply for the calling contract, not `ArbitraryCaller`.

4. The other two options will forward all `msg.value` to `target`. Now `target` can be:
   - Itself, which will result in an infinite loop,
     * but because of gas / bounded call stack, this is not posssible, so state will be reverted.
   - A different contract where somewhere during execution we call back into AC.
     * Since we didn't assume this is the first message call, this can be reduced to:
       · A different contract that doesn't touch AC.
       · Value will be forwarded.