Eggs Finance
Security Review

Cantina Managed review by:

Optimum, Lead Security Researcher
Kaden, Security Researcher

February 11, 2025

Contents

1 Introduction **2**
- 1.1 About Cantina . . . 2
- 1.2 Disclaimer . . . 2
- 1.3 Risk assessment . . . 2
  - 1.3.1 Severity Classification . . . 2

2 Security Review Summary **3**

3 Findings **4**
- 3.1 Critical Risk . . . 4
  - 3.1.1 The contract can be drained of its `sonic` via reentrancy . . . 4
  - 3.1.2 `repay` does not decrease borrowed amount as expected . . . 4
- 3.2 High Risk . . . 5
  - 3.2.1 Leveraged positions can be created before the system `start` . . . 5
- 3.3 Medium Risk . . . 5
  - 3.3.1 Inconsistent interest charged due to integer truncation . . . 5
  - 3.3.2 Users can bypass the borrow expiry restriction by calling `leverage()` instead . . . 6
  - 3.3.3 Floored rounding can cause DoS . . . 6
- 3.4 Low Risk . . . 7
  - 3.4.1 Fees will not be charged if `BUY_FEE` is set low enough . . . 7
  - 3.4.2 `FEE_ADDRESS` is not initialized in the constructor which may lead to loss of fees . . . 7
  - 3.4.3 Missing setters for mutable `SELL_FEE` and `BUY_FEE_REVERSE` . . . 7
- 3.5 Informational . . . 8
  - 3.5.1 `liquidate` calls inside functions will not be effective in case caller's loan is expired . . . 8

1 Introduction

1.1 About Cantina

Cantina is a security services marketplace that connects top security researchers and solutions with clients. Learn more at cantina.xyz

1.2 Disclaimer

Cantina Managed provides a detailed evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While Cantina Managed endeavors to identify and disclose all potential security issues, it cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities that were absent during the initial review. Therefore, any changes made to the code require a new security review to ensure that the code remains secure. Please be advised that the Cantina Managed security review is not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

Severity	Description
Critical	*Must* fix as soon as possible (if already deployed).
High	Leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
Medium	Global losses <10% or losses to only a subset of users, but still unacceptable.
Low	Losses will be annoying but bearable. Applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.
Gas Optimization	Suggestions around gas saving practices.
Informational	Suggestions around best practices or readability.

1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above table. Critical findings have a high likelihood of being exploited and must be addressed immediately. High findings are almost certain to occur, easy to perform, or not easy but highly incentivized thus must be fixed as soon as possible.

Medium findings are conditionally possible or incentivized but are still relatively likely to occur and should be addressed. Low findings a rare combination of circumstances to exploit, or offer little to no incentive to exploit but are recommended to be addressed.

Lastly, some findings might represent objective improvements that should be addressed but do not impact the project's overall security (Gas and Informational findings).

2 Security Review Summary

Eggs Finance is a DeFi protocol built on the Sonic blockchain that utilizes the EGGS token, which is secured by S, to provide a stable and potentially appreciating value mechanism.

From Jan 31st to Feb 2nd the Cantina team conducted a review of eggs on commit hash 28dd9fd2. The team identified a total of **10** issues:

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	2	2	0
High Risk	1	1	0
Medium Risk	3	2	1
Low Risk	3	3	0
Gas Optimizations	0	0	0
Informational	1	1	0
Total	**10**	**9**	**1**

3 Findings

3.1 Critical Risk

3.1.1 The contract can be drained of its `sonic` via reentrancy

Severity: Critical Risk

Context: *(No context files were provided by the reviewer)*

Summary: Unlike other externally accessible, state-changing functions, both the `Eggs.borrow` and `Eggs.borrowMore` functions do not have a `nonReentrant` modifier. In a specific circumstance, the `borrowMore` function can be reentered to drain the contract of its `sonic`.

Finding Description: `Eggs.borrowMore` contains functionality to allow users to borrow more `sonic` without supplying additional collateral in case their position is already sufficiently over-collateralized:

```
uint256 userEggs = SONICtoEGGSNoTrade(sonic);
uint256 userBorrowedInEggs = SONICtoEGGSNoTrade(userBorrowed);
uint256 userExcessInEggs = ((userCollateral) * 99) /
    100 -
    userBorrowedInEggs;

uint256 requireCollateralFromUser = userEggs;
if (userExcessInEggs >= userEggs) {
    requireCollateralFromUser = 0;
} else {
    requireCollateralFromUser =
        requireCollateralFromUser -
        userExcessInEggs;
}

if (requireCollateralFromUser != 0) {
    _transfer(msg.sender, address(this), requireCollateralFromUser);
}
```

Since this function does not have a `nonReentrant` modifier, an attacker can reenter when the additional `sonic` to borrow is sent to them via a `call`:

```
sendSonic(msg.sender, newUserBorrow - sonicFee);
```

```
function sendSonic(address _address, uint256 _value) internal {
    (bool success, ) = _address.call{value: _value}("");
    require(success, "SONIC Transfer failed.");
    emit SendSonic(_address, _value);
}
```

This allows the attacker to call `borrowMore` again with the same parameter. Since `Loans[msg.sender]` hasn't yet been updated, the attacker still doesn't have to provide collateral to borrow the same amount of `sonic`.

We run `safetyCheck` at the end of the function in part to protect the invariant that the price is monotonically increasing:

```
uint256 newPrice = (getBacking() * 1 ether) / totalSupply();
// ...
require(lastPrice <= newPrice, "The price of eggs cannot decrease");
lastPrice = newPrice;
```

Since we increase the `totalBorrowed` via `addLoansByDate` by the amount being transferred to the user, while keeping the actual and stored collateral amounts constant, the price actually remains constant. As a result, the attacker can drain the contract of its entire `sonic` balance via this attack while still passing the `safetyCheck`.

Impact Explanation: An attacker can drain the contract of its entire `sonic` balance. Since `sonic` is the backing of the `eggs` token, the value of `eggs` becomes ~0 as a result.

Likelihood Explanation: The only requirement for this attack to take place is a sufficiently over-collateralized `Loan`. As long as there is activity in the contract, generating fees, loans will naturally become over-collateralized. This will allow for more `sonic` to be borrowed without providing additional collateral, thereby allowing the attack to take place.

Recommendation: Add the `nonReentrant` modifier to both the `borrow` and `borrowMore` functions. Additionally, it's best practice to follow the checks-effects-interations pattern, which would also prevent this by setting storage (effects), like `Loans[msg.sender]`, before making any external calls (interactions).

Eggs Finance: Fixed in commit 2f02fb77.

Cantina Managed: Fixed as recommended.

3.1.2 `repay` does not decrease borrowed amount as expected

Severity: Critical Risk

Context: Eggs.sol#L342

Description: `repay` is a function that allows the borrower to repay a portion of the borrowed funds. The function successfully receives the SONIC payment but do not decrease the corresponding value of `borrowed` in the `Loan` struct as we can see:

```
uint256 newBorrow = borrowed - msg.value;
Loans[msg.sender].borrowed - newBorrow;
```

This error will result in the caller losing these funds.

Recommendation: Short term: consider changing the issued line to:

```
Loans[msg.sender].borrowed = newBorrow;
```

Long term: consider improving test coverage to detect these kind of issues.

Eggs Finance: Fixed in commit 2f02fb77.

Cantina Managed: Fixed as recommended.

3.2 High Risk

3.2.1 Leveraged positions can be created before the system `start`

Severity: High Risk

Context: Eggs.sol#L151-L154

Description: The protocol contains a `start` storage variable, which only the `owner` can set:

```
function setStart() public onlyOwner {
    start = true;
    emit Started(true);
}
```

Before this point, it should not be possible to interact with the system, as is enforced in `buy`:

```
require(start, "Trading must be initialized");
```

Most other state-changing functions do not require this enforcement because the user must hold `eggs` to execute the functions. However, the `leverage` function only requires the user to hold `sonic` and doesn't enforce that `start == true`. As a result, it's possible to create a leveraged position before it's possible to purchase `eggs`. This would give a significantly unfair advantage as they would be able to open a position at the lowest possible price, before anyone else.

Recommendation: Add a `require(start)` check in `leverage` to ensure that leveraged positions cannot be opened before `start == true`.

Eggs Finance: Fixed in commit 2f02fb77.

Cantina Managed: Fixed as recommended.

3.3 Medium Risk

3.3.1 Inconsistent interest charged due to integer truncation

Severity: Medium Risk

Context: Eggs.sol#L209

Description: In `Eggs.getInterestFeeInEggs`, we compute the interest fee with the following logic:

```
uint256 interest = ((3900 * numberOfDays) / 365) + 100;
```

Using `numberOfDays = 1` as an example, we can compare the output of this logic using integers versus real numbers:

- Integers: `3900 * 1 / 365 = 10` (0.01%).
- Real numbers: `3.9% * 1 / 365 ~= 0.0106849315%`.

Here we can see that the output using integers is rounded down by nearly 7%. Since Solidity arithmetic uses integers, this becomes a problem. This can be intentionally exploited by borrowing for shorter durations and repeatedly extending the loan instead of creating a full duration loan. We can see how this creates a variance in the interest paid for one 365 day loans versus 365 one day loans:

- `3900 * 365 / 365 = 3900` (3.9%).
- `(3900 * 1 / 365) * 365 = 3650` (3.65%).

Recommendation: We can represent the interest rate using a larger denominator to get more precision. A good option would be to use 1e18, adjusting the math accordingly. Additionally, we should use a `mulDiv` function to prevent overflow in the intermediary value:

```
- uint256 interest = ((3900 * numberOfDays) / 365) + 100;
+ uint256 interest = mulDiv(0.039e18, numberOfDays, 365) + 0.001e18;
- return ((amount * interest) / 100 / FEE_BASE_1000);
+ return mulDiv(amount, interest, 1e18);
```

Note: The above logic has not been tested.

Eggs Finance: Fixed in commit 2f02fb77.

Cantina Managed: Fixed as recommended.

3.3.2 Users can bypass the borrow expiry restriction by calling `leverage()` instead

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Description: The `leverage()` function acts as a function that combines both `buy()` and `borrow()`. The `borrow()` function restricts user from taking loans for a period of more than a year:

```
require(
    numberOfDays < 366,
    "Max borrow/extension must be 365 days or less"
);
```

However, `leverage()` lacks this kind of check, effectively allowing users to bypass it.

Recommendation: Consider adding this check to `leverage()` as well.

Eggs Finance: Fixed in commit 2f02fb77.

Cantina Managed: Fixed as recommended.

3.3.3 Floored rounding can cause DoS

Severity: Medium Risk

Context: Eggs.sol#L548-L554

Description: We convert between `eggs` and `sonic` based on the current computed price, e.g.:

```
function EGGStoSONIC(uint256 value) public view returns (uint256) {
    return (value * getBacking()) / totalSupply();
}

function SONICtoEGGS(uint256 value) public view returns (uint256) {
    return (value * totalSupply()) / (getBacking() - value);
}
```

In each of these conversion functions, we perform a division to determine the output. In case the resulting value is not a perfect integer, we always round down to the nearest integer.

This can lead to problems in a few circumstances. Let's consider a simple example where we convert from `eggs` to `sonic` and then back:

- `value` = 100 (eggs).
- `getBacking()` = 1000.
- `totalSupply()` = 300.
- `sonic` received = `EGGStoSONIC(value) = (100 * 1000) / 300 = 333`.
- `eggs` received back = `SONICtoEGGS(sonic) = (333 * 300) / (1333 - 333) = 99`.

Above we can see that converting 100 `eggs` to `sonic` and then back leaves us with only 99 `eggs`. In case the last remaining user in the system attempts to close their position, there may not be sufficient `sonic` in the system to cover the computed amount for them to receive, causing a DoS.

Recommendation: It's important that we always round in favor of the protocol. Essentially what this means is that if we're computing an amount for the protocol to receive, we should round up. Conversely, if we're computing an amount for a user to receive, we should round down. We can adjust our logic to round up by adding the `(denominator - 1)` to the numerator, e.g. for `EGGStoSONIC`, that would look something like:

```
// UNTESTED
function EGGStoSONICceil(uint256 value) public view returns (uint256) {
    return (value * getBacking() + (totalSupply() - 1)) / totalSupply();
}
```

The best approach is probably to have separate functions for rounding up versus down for each type of conversion (as necessary), then carefully consider which one should be used at any given time, depending on whether tokens are coming in or out of the protocol.

Eggs Finance: Acknowledged.

Cantina Managed: Partial fix for now since `EGGStoSONICceil` was added but is not used.

3.4 Low Risk

3.4.1 Fees will not be charged if `BUY_FEE` is set low enough

Severity: Low Risk

Context: Eggs.sol#L93

Description: In `Eggs.buy`, we always take a 1/125th fee off the `msg.value` based on the `FEES_BUY` constant:

```
uint256 feeAddressAmount = msg.value / FEES_BUY;
```

There is also a separate `BUY_FEE`, which is mutable, and is taken out of the amount of `eggs` to `mint` :

```
mint(reciever, (eggs * getBuyFee()) / FEE_BASE_1000);
```

In case `BUY_FEE` ever corresponds to a fee less than 1/125th, the user will effectively not be paying that fee since they will receive the full amount of `eggs` regardless.

Recommendation: Currently, it's possible to set `BUY_FEE` to any value between 1000 and 975. Instead, we should limit the maximum to correspond to a 1/125th fee, which would be 992:

```
  function setBuyFee(uint16 amount) external onlyOwner {
-     require(amount <= 1000, "buy fee must be less 0% or more");
+     require(amount <= 992, "buy fee must be at least 0.8%");
      require(amount >= 975, "buy fee must be less than 2.5%");
      BUY_FEE = amount;
      emit buyFeeUpdated(amount);
  }
```

Note that this also affects `sell` and assuming a function is added to set the `SELL_FEE`, we should enforce the same maximum.

Eggs Finance: Fixed in commit c177bc97.

Cantina Managed: Fixed as recommended.

3.4.2 `FEE_ADDRESS` is not initialized in the constructor which may lead to loss of fees

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: The initialization of the contract consists of two main functions; the constructor and `setStart()`. Functions like `buy()` (and others) can not be called before `setStart()` is called, but they can be called while `FEE_ADDRESS` is not initialized (by calling `setFeeAddress()`) leading to lost fees.

Recommendation: Consider initializing `FEE_ADDRESS` either in the constructor or in `setStart()`.

Eggs Finance: Fixed in commit 2f02fb77 by reverting in `setStart` if `FEE_ADDRESS == address(0x0)`.

Cantina Managed: Verified.

3.4.3 Missing setters for mutable `SELL_FEE` and `BUY_FEE_REVERSE`

Severity: Low Risk

Context: Eggs.sol#L13-L15

Description: We have the following mutable fee storage variables:

```
uint16 public SELL_FEE = 975;
uint16 public BUY_FEE = 975;
uint16 public BUY_FEE_REVERSE = 10;
```

Of these, only `BUY_FEE` has a setter (`setBuyFee`). If it's intended for `SELL_FEE` and `BUY_FEE_REVERSE` to be mutable, they should have setter functions. Alternatively, if they should be immutable then we can set them as constants to save a significant amount of gas each time they're used.

Recommendation: Either implement setter functions for `SELL_FEE` and `BUY_FEE_REVERSE` or make them constants.

Note: If you choose to add setter functions, carefully consider the minimum and maximum values to be enforced, as recommend in the other finding: Fees will not be charged if `BUY_FEE` is set low enough.

Eggs Finance: Fixed in commit c177bc97.

Cantina Managed: Fixed as recommended.

3.5 Informational

3.5.1 `liquidate` calls inside functions will not be effective in case caller's loan is expired

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: The protocol implements loans that expire on a specific date. After the expiry date, loans that were not closed will be liquidated upon the next call to `liquidate`:

```
function liquidate() public {
    uint256 borrowed;
    uint256 collateral;

    while (lastLiquidationDate < block.timestamp) {
        collateral += CollateralByDate[lastLiquidationDate];
        borrowed += BorrowedByDate[lastLiquidationDate];
        lastLiquidationDate += 1 days;
    }

    if (collateral != 0) {
        totalCollateral -= collateral;
        _burn(address(this), collateral);
    }

    if (borrowed != 0) {
        totalBorrowed -= borrowed;
        emit Liquidate(lastLiquidationDate - 1 days, borrowed);
    }
}
```

To make the system more efficient, this function is being as part of any function that modify the state. However, functions like `borrowMore`, `removeCollateral`, `repay`, `closePosition`, `flashClosePosition`, and `extendLoan` will revert in case the function caller has an expired loan, which means the effects of `liquidate` will also revert.

Recommendation: Consider implementing an off-chain script that will call `liquidate` on a daily basis as well as placing the expiration check for the caller loan before the call to `liquidate`. Another option which is less recommended will be to return early instead of reverting in case the loan has expired.

Eggs Finance: Fixed in commit 2f02fb77.

Cantina Managed: Fixed as recommended.